Ejecución eficiente de secuencias de navegación web

Autor: José Losada Pérez

Tesis doctoral UDC / 2015

Directores:

Juan Raposo Santiago

Carlos Alberto Pan Bermúdez

Programa Oficial de Doutoramento en Tecnoloxías da Información e as Comunicacións

UNIVERSIDADE DA CORUÑA

D. **Juan Raposo Santiago** y D. **Carlos Alberto Pan Bermúdez**, Doctores por la Universidade da Coruña

CERTIFICAN

Que la presente memoria titulada **Ejecución eficiente de secuencias de navegación web,** ha sido realizada bajo su dirección y constituye la Tesis que presenta D. **José Losada Pérez** para optar al grado de Doctor por la Universidade da Coruña.

A Coruña, 2015

D. Juan Raposo Santiago D. Carlos Alberto Pan Bermúdez

AGRADECIMIENTOS

Me gustaría dar las gracias de una manera especial a los directores de esta tesis, Juan y Alberto, por toda su ayuda y su apoyo a lo largo de estos años.

También me gustaría acordarme del resto de mis compañeros de departamento, siempre dispuestos a echar una mano.

Y gracias a mi familia, que siempre está ahí, y a Eva, que ha tenido la valentía de leer este documento.

RESUMEN

Las aplicaciones de automatización web se utilizan ampliamente para diferentes propósitos, tales como integración B2B o pruebas automatizadas de aplicaciones web.

En la mayoría de los sistemas, el componente de navegación automático, se desarrolla utilizando las APIs de navegadores convencionales (ej. Internet Explorer o Firefox). Esta aproximación, presenta problemas de rendimiento en tareas de automatización web intensivas, que requieren de respuestas en tiempo real y/o de un alto nivel de paralelismo.

Otros sistemas, utilizan componentes de navegación especializados, que omiten ciertas etapas de procesamiento de los navegadores convencionales, pero funcionan de forma similar a éstos, a la hora de cargar y construir la representación interna de las páginas web.

En esta tesis doctoral, se presenta un conjunto de técnicas de optimización, que permiten ejecutar eficientemente tareas de automatización web. Estas técnicas, se aplican cuando el componente de navegación construye la representación interna de las páginas web.

Además, también se ha diseñado una arquitectura completa para un componente de navegación especializado en automatización web, que da soporte a las novedosas técnicas de optimización diseñadas previamente.

Para validar experimentalmente las técnicas de optimización y el rendimiento de la arquitectura, se ha desarrollado una implementación de referencia, que sigue todos los principios de esta arquitectura. Dicha implementación, ha sido comparada con otros sistemas de navegación, utilizando sitios web reales.

ABSTRACT

Web automation applications are widely used for different purposes, such as B2B integration and automated testing of web applications.

Most current systems, build the automatic web navigation component by using the APIs of conventional browsers. This approach suffers performance problems for intensive web automation environments, which require real time responses and/or a high degree of parallelism.

Other systems, use the approach of creating custom browsers to avoid some of the processing steps of conventional browsers, but they work like them when loading and building the internal representation of the web pages.

In this PhD, we present a set of optimization techniques that allow an efficient execution of the Web automation tasks. These techniques, can be applied when the navigation component builds the internal representation of the Web page.

In addition, we designed a complete architecture for a custom browser specialized in the execution of Web automation tasks. The proposed architecture, supports the novel optimization techniques previously designed.

Finally, to evaluate the validity of the optimization techniques and the performance of the proposed architecture, we develop a reference implementation following the architecture principles. This reference implementation, was validated comparing it with other navigation components, using real Web sites.

RESUMO

As aplicacions de automatización web, utilízanse amplamente para diferentes propósitos, tales como integración B2B ou probas automatizadas de aplicacións web.

Na meirande parte dos sistemas, o compoñente de navegación automático desenrólase empregando as APIs dos navegadores convencionais (ex. Internet Explorer ou Firefox). Esta aproximación, presenta problemas de rendemento en tarefas de automatización web intensivas que requiren de respostas en tempo real e/ou de un alto grao de paralelismo.

Outros sistemas, empregan compoñentes de navegación especializados, que omiten certas etapas de procesamento dos navegadores convencionais, pero funcionan de maneira similar a estes, á hora de cargar e construír a representación interna da páxina web.

Nesta tese de doutoramento, preséntase un conxunto de técnicas de optimización, que permiten executar eficientemente, tarefas de automatización web. Estas técnicas, aplícanse cando o compoñente de navegación constrúe a representación interna das páxinas web.

Ademais, tamén se deseñou unha arquitectura completa para un compoñente de navegación especializado na automatización web, que da soporte as novidosas técnicas de optimización deseñadas previamente.

Para validar experimentalmente as técnicas de optimización e o rendemento da arquitectura, desenrolouse unha implementación de referencia, que segue todos os principios desta arquitectura. Dita implementación, foi comparada con outros sistemas de navegación, empregando sitios web reais.

Índice de contenidos

Índice de figuras

Índice de tablas

1 PLANTEAMIENTO Y CONTRIBUCIONES

1.1 ÁMBITO

1.1.1 La Web de hoy y las aplicaciones de automatización web

La World Wide Web (WWW), también conocida como la Web, es un sistema de obtención y presentación de documentos distribuidos por Internet. Estos documentos, pueden incluir cualquier tipo de contenido (texto, imágenes, vídeos, etc.) y además, se encuentran conectados entre sí a través de "hiperenlaces".

Hoy en día, la Web se ha convertido en el mayor centro de información, comunicación, interacción y servicios jamás construido, con dimensión y acceso mundial.

A la WWW se accede, en gran medida, a través de un programa llamado navegador web, cuya funcionalidad básica es la de permitir la visualización de los documentos descargados.

La Web, ha sido diseñada para ser utilizada por personas, por lo que la mayoría de los sitios web accesibles a través de Internet, no proporcionan interfaces programáticas para que otros programas puedan interactuar con ellos, de forma similar a cómo lo hace un usuario humano, a través de un navegador web.

Por ello, en los últimos años, ha aumentado el interés en las llamadas aplicaciones de automatización web. Estas aplicaciones, permiten la interacción automática con los sitios web, reproduciendo las acciones que un usuario humano ejecuta sobre un navegador convencional.

Las aplicaciones de automatización web, se utilizan en diferentes ámbitos, entre los que destacan los siguientes: integración B2B, web mashups, pruebas automatizadas de aplicaciones web, aplicaciones de meta-búsqueda en Internet o aplicaciones de vigilancia tecnológica.

Por ejemplo, una aplicación de automatización web de vigilancia tecnológica dentro del ámbito de la investigación, podría ejecutarse todos los días para: en primer lugar, acceder a diferentes sitios web de búsqueda de patentes y artículos, a continuación, ejecutar de manera automática la misma búsqueda en todos ellos (por ejemplo, para obtener los artículos y patentes de un área de conocimiento específico), y en último lugar, extraer todos los resultados y mostrarlos ordenados por algún criterio predefinido.

1.1.2 Las secuencias de navegación web automática

Una parte fundamental dentro de las tecnologías de automatización web, es la capacidad para ejecutar las llamadas secuencias de navegación web.

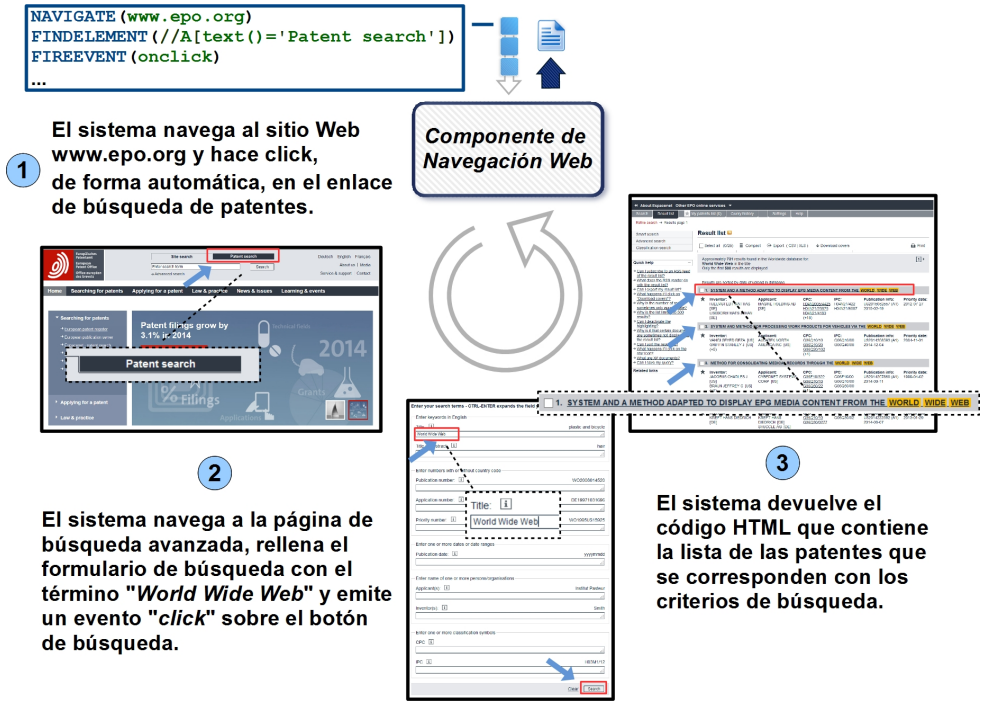


Figura 1. Ejemplo de secuencia de navegación web.

Una secuencia de navegación web, consiste en una sucesión de comandos que se ejecutan de forma automática sobre un componente de navegación especializado y donde cada uno de estos comandos, representa una acción que un usuario humano podría ejecutar sobre un navegador convencional.

La Figura 1, muestra un ejemplo de una secuencia de navegación web, que obtiene información de patentes de la página web de la Oficina Europea de Patentes (www.epo.org).

Para ello, en primer lugar, la secuencia de navegación accede a la página inicial del sitio web y a continuación, ejecuta un evento "*click*" sobre el hiperenlace que contiene el texto "*Patent search*".

Una vez que se ha pulsado sobre este primer enlace, es posible acceder a la página de búsqueda avanzada de patentes pulsando en otro hiperenlace, en este caso, se trata del hiperenlace que contiene el texto "*Advanced patent search*" (este segundo enlace, se ha hecho visible una vez que ha finalizado la ejecución del primero de los eventos).

Este segundo hiperenlace, navega a la página de búsqueda avanzada, en la que se rellena el campo de texto con el término "*Word Wide Web*" y a continuación se pulsa en el botón de buscar.

Tras la ejecución de la secuencia, el sistema de automatización web devuelve el código HTML [W3CHTML4], con la lista de patentes obtenidas según el criterio de búsqueda.

La Figura 2, muestra la secuencia de navegación web del ejemplo de la Figura 1, en un sistema de automatización web en particular [RPAHV02].

```
1: NAVIGATE(www.epo.org)
2: FINDELEMENT(//A[text()='Patent search'])
3: FIREEVENT(onclick)
4: FINDELEMENT(//BUTTON[text()='Advanced patent search'))
5: FIREEVENT(onclick)
6: FINDELEMENT(//TEXTAREA[@name='TI'])
7: SETVALUE(Word Wide Web)
8: FINDELEMENT(//INPUT[@value='Search'])
9: FIREEVENT(onclick)
```

Figura 2. Comandos de la secuencia de navegación web del ejemplo.

En la figura, se pueden observar algunos de los comandos más representativos y que habitualmente forman parte de las secuencias de navegación en este sistema de automatización web (en otros sistemas, los comandos son similares, variando la sintaxis específica de cada uno de ellos).

Entre estos comandos, destacan los siguientes:

1. El comando *Navigate*, utilizado para navegar a una página web, identificada mediante una URL.
2. El comando *FindElement*, utilizado para localizar y seleccionar un elemento en la página HTML, por medio de una expresión XPath [XPATH] que identifica de manera única al elemento dentro del documento.
3. El comando *FireEvent,* que ejecuta un evento sobre el último elemento seleccionado mediante un comando de selección (como por ejemplo, *FindElement* mencionado con anterioridad).
4. O el comando *SetValue*, que establece un valor sobre el último elemento seleccionado (este comando se utiliza, sobre todo, en campos de entrada de formularios para establecer valores de búsqueda, antes de enviar el formulario).

Una misma secuencia de navegación web, se puede ejecutar múltiples veces sobre el mismo sitio web y es habitual que se puedan especificar diferentes parámetros de búsqueda, en cada una de las ejecuciones.

La Figura 3, detalla la misma secuencia del ejemplo de la Figura 2, pero en este caso, en vez de ejecutar siempre la misma búsqueda sobre el sitio web de la Oficina Europea de Patentes, esta secuencia permite modificar la ejecución, especificando mediante el parámetro *@query* un término de búsqueda diferente, en cada una de las ejecuciones.

```
1: NAVIGATE(www.epo.org)
2: FINDELEMENT(//A[text()='Patent search'])
3: FIREEVENT(onclick)
4: FINDELEMENT(//BUTTON[text()='Advanced patent search'))
5: FIREEVENT(onclick)
6: FINDELEMENT(//TEXTAREA[@name='TI'])
7: SETVALUE(@query)
8: FINDELEMENT(//INPUT[@value='Search'])
9: FIREEVENT(onclick)
```

Figura 3. Secuencia de navegación web con parámetros.

Así pues, con esta misma secuencia de navegación, se podrían ejecutar diferentes búsquedas sobre el mismo sitio web, especificando diferentes valores para los parámetros de entrada (por ejemplo: "*Word Wide Web*" o "*Internet*").

1.1.3 Los componentes de navegación web

Para la ejecución de secuencias de navegación web, se utiliza un software especializado, conocido como componente de navegación web.

Para construir estos componentes, la aproximación que siguen la mayoría de los sistemas de automatización web existentes hoy en día, consiste en la utilización de las API de alto nivel que proporcionan los principales navegadores convencionales.

Esta aproximación, tiene la ventaja de que evita el desarrollo de un motor de navegación desde cero y garantiza que las páginas web a las que se accede, se comportarán exactamente igual que cuando un usuario humano accede a ellas, utilizando ese mismo navegador.

El principal inconveniente de esta aproximación es que los navegadores web convencionales son aplicaciones cliente, que no han sido desarrollados pensando en tareas de automatización web, por lo que el rendimiento puede no ser el ideal en tareas de automatización que requieren un alto número de navegadores ejecutándose de manera concurrente, o cuando la tarea requiere una respuesta en tiempo real.

Por el contrario, otros sistemas de automatización, siguen una aproximación diferente, que consiste en desarrollar navegadores a medida (ad-hoc) especializados en la ejecución de estas secuencias de navegación web.

Un inconveniente de estos sistemas, es que su implementación es una tarea muy costosa, en la que es necesario dar soporte a todas las funcionalidades disponibles en un navegador convencional, algunas de ellas de considerable complejidad. Pero por otra parte, tienen la ventaja de ser más eficientes, dado que no están pensados para ser utilizados por personas y pueden evitar algunas de las tareas que realizan los navegadores convencionales para mostrar el contenido descargado de la Web.

A pesar de ser más eficientes que los navegadores convencionales, el modelo de funcionamiento de los componentes a medida existentes hasta la fecha, es muy similar al de los navegadores convencionales, por lo que, en muchos casos, la

mejora de rendimiento no es demasiado significativa, y la cantidad de recursos que consumen aún es bastante elevada.

En este trabajo, se aborda la tarea de ejecutar las secuencias de navegación web de la forma más eficiente posible, en el contexto de los navegadores ad-hoc.

Para ello, en una primera fase, se han diseñado una serie de técnicas y algoritmos de optimización, basados en diversas características y peculiaridades de los entornos de automatización web.

La primera de esas características, es que la misma secuencia de navegación, se ejecuta múltiples veces sobre el mismo sitio web, con lo que es posible extraer cierta información en una ejecución inicial, para utilizar dicha información, con posterioridad, en las subsiguientes ejecuciones de la misma secuencia.

Y la segunda, ya comentada previamente, es que la interfaz de usuario no es necesaria, por lo que la información de visualización podría ser calculada parcialmente y sólo cuando es estrictamente necesaria. Hay que tener en cuenta que los componentes de navegación ad-hoc actuales, aunque no disponen de interfaz gráfica, calculan igualmente toda la información de visualización, por si ésta es necesaria para la correcta ejecución de la secuencia (es necesaria, cuando se accede a ella desde código de scripting contenido en la página).

En una segunda fase, se ha diseñado la arquitectura completa de un componente de navegación web, en la cual se han tenido en cuenta todos los principios y técnicas de optimización de la fase anterior. Como se verá en el capítulo 5, la aplicación de estas técnicas, mejora notablemente la eficiencia de las tareas de automatización web.

1.2 OBJETIVOS

En esta tesis doctoral, se estudia la ejecución eficiente de secuencias automáticas de navegación web, dentro del contexto de los navegadores web ad-hoc.

Para ello, el problema se ha dividido en dos fases. Una primera en la que se han diseñado una serie de técnicas de optimización que aprovechan las características específicas de los entornos de automatización web y una segunda, en la que se ha diseñado una arquitectura para un componente de navegación web optimizado, capaz de dar soporte a esas técnicas y algoritmos.

Previamente al desarrollo de estas técnicas, se ha realizado un estudio del estado del arte, analizando otros trabajos sobre automatización web y analizando también el modo de funcionamiento y las diferentes técnicas de optimización desarrolladas en los principales navegadores convencionales.

Además, se presenta una implementación de referencia de la arquitectura del componente de navegación, que da soporte a las técnicas propuestas. Esta implementación de referencia, ha sido incorporada a un software comercial de automatización web utilizado en numerosas aplicaciones industriales, lo que contribuye a reforzar la aplicabilidad de las aportaciones de este trabajo.

En último lugar, la arquitectura y las técnicas desarrolladas, se han validado de forma experimental utilizando sitios web reales.

Los objetivos detallados de este trabajo son los siguientes:

1. Proponer distintas técnicas de optimización aplicables en la ejecución de secuencias de navegación web. Estas técnicas están basadas en los siguientes principios:

    a) La misma secuencia de navegación se ejecuta múltiples veces, lo que permite extraer información relevante en una primera ejecución, para utilizar esta información con posterioridad, en las siguiente ejecuciones de la misma secuencia.

    b) Además, por las características particulares de los componentes de navegación web a medida, diseñados de manera específica para la ejecución de secuencias de navegación, algunas funcionalidades que son imprescindibles en los navegadores convencionales, en estos navegadores a medida pueden ser simuladas y en muchos casos optimizadas.

2. Proponer una arquitectura completa de un componente de navegación web que dé soporte a las técnicas de optimización diseñadas en la fase anterior. Además, se realizará una implementación de referencia de la arquitectura desarrollada. La arquitectura del navegador a medida incluirá las siguientes características:

a) Funcionalidades comunes en todos los navegadores convencionales (procesado de documentos HTML y XML, soporte del lenguaje JavaScript, etc.).

b) Soporte para las técnicas de optimización automática diseñadas en la primera fase.

c) Soporte para otras técnicas de optimización manual, que permitirán establecer una configuración específica en el componente de navegación web en cada una de las secuencias. La configuración manual, requiere de la intervención de un usuario con conocimiento de bajo nivel sobre el funcionamiento del sitio web al que se está accediendo.

3. Validar las técnicas propuestas utilizando casos de estudio reales. Cada una de las técnicas de optimización automática que se proponen, debe ser validada por separado midiendo su efectividad, tanto la mejoría obtenida en el tiempo de ejecución, como en el uso de recursos computacionales (por ejemplo, uso de memoria y uso de la red).

4. Validar la arquitectura completa del componente de navegación. En este caso, debe compararse el tiempo de ejecución del navegador web desarrollado a medida, con el tiempo de ejecución de otros componentes de navegación. Para que este estudio comparativo sea lo más completo posible, se incluirán en él los siguientes sistemas:

    a) Un componente de navegación que implementará la arquitectura de referencia propuesta en este trabajo y que utilizará todas las capacidades y técnicas de optimización automática.

    b) Un componente de navegación que implementará la arquitectura de referencia propuesta en este trabajo pero sin utilizar las capacidades y técnicas de optimización automática. En este caso, la implementación de referencia trabajará de manera muy similar a cómo lo hace cualquier otro navegador web ad-hoc.

    c) Además, se incluirá en el estudio otro componente de navegación a medida. Se utilizará en este caso, un navegador web ad-hoc muy conocido y de código abierto.

    d) Por último, se incluirá un tercer componente de navegación que en este caso, utiliza el API de alto nivel de uno de los principales navegadores convencionales.

1.3 PRINCIPALES CONTRIBUCIONES ORIGINALES

Este trabajo presenta las siguientes aportaciones originales:

1. Una técnica de optimización que permite identificar y descartar, de manera automática, los elementos de una página web que no son necesarios para el correcto funcionamiento de una secuencia de navegación web.

   Algunos ejemplos de elementos descartados de forma sistemática son: anuncios de publicidad, marcos de conexión con las distintas redes sociales, JavaScript de generación de estadísticas del tráfico del sitio web, etc.

   También se identifica e ignora cualquier otro tipo de elemento de la página web, que no se utiliza durante la ejecución de la secuencia de navegación: enlaces a páginas a las que no se accede, ficheros JavaScript cuya ejecución es irrelevante, menús desplegables, barras de herramientas laterales, etc.

   Estos elementos irrelevantes, deben ser identificados durante una primera fase, generando la información necesaria para construir, de manera posterior, documentos HTML más pequeños que sólo incluirán los elementos imprescindibles para el correcto funcionamiento de la secuencia.

   Esta técnica de optimización, se describe en el apartado 3.1.

2. Una técnica de optimización que permite ejecutar código JavaScript en paralelo, durante la construcción de un documento HTML.

   Esta técnica, está basada en un análisis preliminar de las dependencias durante la ejecución de los scripts incluidos en un documento HTML. Tras ese análisis, será posible determinar qué scripts se pueden ejecutar de manera simultánea, sin que ello produzca un error en la construcción de la página web (y por lo tanto, un error en la ejecución de la secuencia de navegación).

   Esta técnica de optimización, se describe en el apartado 3.2.

3. Una técnica de optimización que permite omitir el cálculo de la información de visualización, en los casos en los que no es necesaria para un navegador ad-hoc.

   Los navegadores a medida, desarrollados para tareas de automatización web, no son aplicaciones cliente pensadas para ser utilizadas por personas, por lo que no necesitan interfaz de usuario.

   Por lo tanto, el proceso de renderización de la página web puede no realizarse, con lo que el cálculo de la información de presentación, tampoco sería necesario, salvo cuando esta información es accedida durante la ejecución del código JavaScript.

   Esta técnica de optimización, se describe en el apartado 3.3.

4. Una arquitectura completa para un componente de navegación web ad-hoc, que da soporte a las técnicas de optimización automática propuestas en los tres puntos anteriores.

   Se proporciona, además, una implementación de referencia de esta arquitectura. Esta implementación de referencia, se puede ejecutar utilizando las capacidades de optimización propuestas con anterioridad, o se puede ejecutar sin ellas, actuando en este caso de forma similar a cómo lo hacen el resto de navegadores desarrollados a medida.

   Esta arquitectura, se describe en el capítulo 4.

Las técnicas de optimización propuestas y la arquitectura del componente de navegación, han sido validadas de manera experimental, utilizando casos de estudio reales y han sido incorporadas a un software comercial de automatización web, utilizado en numerosas aplicaciones reales y proyectos de automatización web de diversos ámbitos, soportando requerimientos y carga de trabajos reales.

Más concretamente, la implementación de referencia de la arquitectura del componente de navegación con soporte para las diferentes técnicas de optimización, se incluye dentro de un módulo de automatización web (ITPilot [DNDITP]), que forma parte de la Plataforma Denodo [DNDPLATF], una suite de integración de datos distribuidos en tiempo real, desarrollada y explotada de forma comercial por Denodo Technologies, compañía internacional con reconocido prestigio a nivel mundial en las áreas de la virtualización de datos y la automatización web.

1.4 ESTRUCTURA DE LA TESIS

El capítulo 2, "*Estado del arte*", comienza con una breve contextualización de las aplicaciones de automatización web, su motivación y las principales soluciones existentes hoy en día.

En este capítulo, también se describe la arquitectura de los principales navegadores convencionales. Los navegadores convencionales, han sido utilizados como base inicial para construir la arquitectura de navegación desarrollada en esta tesis doctoral.

Por último, en este capítulo se describen los fundamentos teóricos que soportan las distintas técnicas de optimización propuestas.

El capítulo 3, "*Descripción de las técnicas de optimización para la ejecución eficiente de secuencias de navegación web*", detalla de manera exhaustiva, las técnicas de optimización propuestas en esta tesis doctoral.

El capítulo 4, "*Descripción de la arquitectura de un componente de navegación optimizado para la ejecución eficiente de secuencias de navegación web*", describe la arquitectura del componente de navegación a medida con soporte para las técnicas de optimización detalladas en el capítulo 3.

El capítulo 5, "*Experiencia obtenida con el uso del sistema*", detalla todos los experimentos que han sido desarrollados para validar, tanto cada una de las técnicas de optimización por separado, como la arquitectura completa del sistema de navegación. En este último caso, se ha comparado la implementación de referencia del navegador a medida, con otros sistemas de automatización web que también han sido incluidos en los experimentos.

Con posterioridad, en este mismo capítulo, se analiza la utilización del componente de navegación dentro de un producto comercial real y en último lugar, se realiza una evaluación del cumplimiento de los objetivos planteados, en función de los datos recogidos de los experimentos.

En el capítulo 6, "Discusión, conclusiones y líneas de trabajo futuro", se discuten los resultados y aportaciones originales de este trabajo, en comparación con los trabajos previos descritos en el capítulo 2, a continuación, se exponen las conclusiones de la tesis doctoral y en último lugar, se esbozan las líneas de trabajo futuro del autor.

2 ESTADO DEL ARTE

En este capítulo, se contextualizan las aplicaciones de automatización web y para ello, en primer lugar, se muestra una visión general de dónde surge su necesidad y se detalla el estado del arte de las principales soluciones existentes en la actualidad.

Así mismo, en este capítulo también se presenta una visión general de los fundamentos teóricos que han sido utilizados para diseñar las distintas técnicas de optimización presentadas en esta tesis doctoral.

Así pues, en primer lugar, en el apartado 2.1, se describe el origen y la necesidad de las aplicaciones de automatización web.

A continuación, en el apartado 2.2, "*Navegadores web convencionales*", se describe la arquitectura y los principios de funcionamiento de los navegadores convencionales más utilizados hoy en día.

En el apartado 2.3, "Automatización web basada en navegadores tradicionales", se detallan las principales soluciones de automatización web basadas en la utilización del API de alto nivel, proporcionado por los navegadores convencionales

A continuación, en el apartado 2.4, "*Navegadores web a medida*", se describe la arquitectura general de los navegadores web a medida, se explican las diferencias con los navegadores convencionales, y se detallan sus principales ventajas e inconvenientes.

En el apartado 2.5 "Automatización web basada en navegadores desarrollados a medida", se detallan las principales soluciones de automatización web basadas en el desarrollo de navegadores a medida.

Y por último, en el apartado 2.6, "*Discusión y conclusiones*", se discuten las ventajas e inconvenientes de las diferentes aproximaciones utilizadas en la actualidad para construir componentes de automatización web y se analizan algunos puntos en los cuales es posible desarrollar mejoras y optimizaciones sobre los sistemas actuales de automatización web que proporcionan componentes de navegación desarrollados a medida.

2.1 INTRODUCCIÓN

Un alto porcentaje de los sitios web de Internet, proporcionan formularios de búsqueda que permiten ejecutar consultas sobre una base de datos subyacente, mostrando los resultados obtenidos en formato HTML.

La mayor parte de estos sitios web, no ofrecen ningún otro tipo de API de acceso a esos datos, como por ejemplo un API de Servicios web que pueda ser consultado de forma programática, sin intervención humana.

Es por ello que, en los últimos años, está creciendo el uso de aplicaciones de automatización web. Estas aplicaciones, permiten el acceso a los datos de los sitios web de manera automática, desarrollando componentes de navegación web que simulan el comportamiento de un usuario humano cuando accede a un sitio de Internet, a través de un navegador web convencional.

El ámbito de utilización de las aplicaciones de automatización web es muy variado, entre otros, destacan los siguientes:

1. Aplicaciones de integración B2B.
2. Web mashups.
3. Pruebas automatizadas de aplicaciones web
4. Aplicaciones de vigilancia tecnológica.

Por ejemplo, si una empresa está interesada en conocer el precio de los productos de sus competidores, podría extraer esa información utilizando una aplicación de automatización web que navegase de manera periódica, a los sitios web de las otras empresas para extraer la información requerida.

Para las aplicaciones de automatización, se utilizan los denominados componentes de navegación web. Estos componentes, basan su funcionamiento en la ejecución de secuencias de navegación web, compuestas éstas por una sucesión de pasos o comandos que se ejecutan de forma secuencial y que simulan el comportamiento de un usuario humano manipulando un navegador convencional.

Para el desarrollo de un componente de navegación web, existen dos aproximaciones diferentes. La primera de ellas, consiste en la utilización de un API de alto nivel de alguno de los principales navegadores web convencionales.

Por ejemplo, el navegador web Mozilla Firefox [MOZFF] ofrece un extenso API de alto nivel, con un gran número de funcionalidades, que posibilitan la implementación de una extensión (*plugin*) para la ejecución de secuencias de navegación web.

Entre estas funcionalidades de alto nivel, podríamos destacar las siguientes:

1. Navegar de forma automática a un sitio web.
2. Controlar el estado de las navegaciones en curso.

3. Manipular el árbol DOM de la página que tiene cargada el navegador.
4. Emitir eventos sobre los elementos del árbol DOM de la página.
5. Manipular sockets y ficheros de disco, etc.

De forma similar, el navegador web Microsoft Internet Explorer [MSIE], también ofrece un API de alto nivel que permite incrustar un componente de navegación, dentro de una aplicación software desarrollada en alguno de los lenguajes soportados en el sistema operativo Microsoft Windows, por ejemplo, en C++. Sobre este componente, también es posible emitir eventos, ejecutar navegaciones, etc.

Esta primera estrategia para construir un componente de navegación web, presenta la ventaja de que es la solución más sencilla y con menos complejidad.

Además, cuando se dispara un evento de manera programática sobre un componente de navegación basado en Firefox o Internet Explorer, está garantizado que el componente se va a comportar exactamente igual que cuando un usuario humano emite el mismo evento sobre el navegador (por ejemplo, pulsar una tecla o mover el ratón).

Por el contrario, esta aproximación presenta importantes problemas de rendimiento cuando las tareas de automatización web requieren de respuestas en tiempo real o cuando es necesario un elevado nivel de concurrencia, con un gran número de navegadores ejecutándose al mismo tiempo.

Cabe recordar que los navegadores web tradicionales no han sido desarrollados pensando en tareas de automatización web, sino que han sido concebidos para ser utilizados por personas, como aplicaciones de escritorio y es por ello que durante su implementación, muchas veces no se da prioridad a la eficiencia y la optimización de recursos computacionales sobre otras características.

Por este motivo, aparece una segunda aproximación a la hora de construir un componente de navegación web para ejecutar secuencias de navegación. Esta alternativa, consiste en desarrollar un navegador a medida desde cero, utilizando como base la arquitectura de los navegadores tradicionales y teniendo en cuenta aspectos específicos de rendimiento y eficiencia para las tareas de automatización web.

Esta segunda aproximación, derivará en componentes más eficientes, al ser concebidos de manera específica, para la ejecución de secuencias de navegación, aunque presenta el gran inconveniente de su desarrollo, dado que para su correcto funcionamiento, es necesario dar soporte a las principales funcionalidades de los navegadores tradicionales, entre las que destacan las siguientes:

1. Procesado de documentos HTML y XML.
2. Soporte para lenguajes interpretados como JavaScript.
3. Soporte de todos los objetos nativos del navegador.

A continuación, en el siguiente apartado, se describe la arquitectura y los principios de funcionamiento de los navegadores convencionales más utilizados hoy en día y a continuación, se detallan las principales soluciones de automatización web basadas en la utilización de estos navegadores convencionales.

2.2 NAVEGADORES WEB CONVENCIONALES

2.2.1 Principales modelos y lenguajes

2.2.1.1 Document Object Model (DOM)

El modelo DOM o *Document Object Model* [W3CDOM] es una convención que permite representar los objetos en los documentos HTML y XML y es uno de los conceptos más importantes utilizados durante todo el desarrollo de este trabajo.

El modelo DOM, describe cómo los navegadores representan las páginas web que tienen cargadas y cómo estas páginas deben responder a los eventos que en ellas se generan.

Cada página HTML se modela como un árbol, en el que cada elemento HTML se representa con el tipo apropiado de nodo.

Una clase de nodos muy importantes son los nodos script, utilizados para almacenar contenido ejecutable. Este código ejecutable, se implementa en algún lenguaje interpretado, como por ejemplo JavaScript.

Los nodos script pueden incluir directamente el código ejecutable o pueden hacer referencia a un fichero externo que contiene ese código.

Estos scripts, se descargan y ejecutan a medida que van apareciendo durante el procesado incremental del documento HTML, y pueden contener declaraciones de elementos (por ejemplo, funciones o variables) que pueden ser utilizados, con posterioridad, en otros scripts o en manejadores de eventos de los nodos de la página.

Todos los nodos contenidos en el árbol DOM de la página, pueden recibir eventos, producidos de manera directa o indirecta por acciones del usuario.

Los diferentes tipos de eventos, existen para acciones como: pulsar sobre un elemento de la página (*click*), mover el ratón sobre un elemento (*mouseover*) o indicar que una página se ha cargado de forma correcta (*load*), por nombrar algunos ejemplos relevantes.

Cada nodo puede registrar un conjunto de manejadores para procesar los diferentes tipos de eventos que puede recibir.

Un manejador de eventos, puede ejecutar código JavaScript arbitrario. En muchos casos, este manejador realiza una invocación a una función, declarada con anterioridad en alguno de los nodos script ejecutados durante el proceso de carga de la página HTML.

Este código de scripting, que se ejecuta para procesar los eventos, puede acceder a todo el árbol DOM de la página y puede ejecutar acciones como por ejemplo, modificar o eliminar nodos existentes, crear nuevos nodos y añadirlos al árbol DOM, o incluso disparar nuevos eventos.

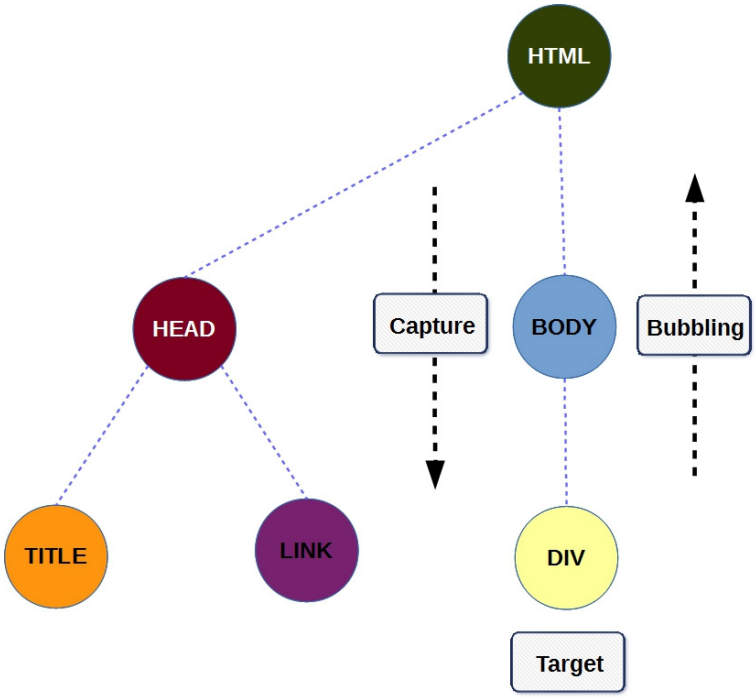


Figura 4. Ciclo de vida durante el procesamiento de un evento.

El ciclo de vida para el procesamiento de un evento, consta de varias fases en las que el evento puede ser gestionado, tanto en el nodo de destino, como en cualquiera de sus ancestros en el árbol DOM de la página.

El proceso se puede resumir de la siguiente manera:

1. En primer lugar, el evento se propaga siguiendo una ruta que comienza en el nodo raíz del árbol DOM y que llega hasta el nodo destino. Esta fase se denomina *capture*.
2. A continuación, el evento se procesa de manera local en este nodo. Esta fase se denomina *target*.
3. En último lugar, el evento se propaga de nuevo siguiendo una ruta que vuelve desde el nodo de destino hasta el nodo raíz. Esta fase se denomina *bubble*.

Los manejadores instalados en un nodo se pueden registrar para las fases de *capture* y de *bubbling*.

Este ciclo de vida en el procesamiento de un evento, es un compromiso entre las dos aproximaciones tradicionalmente más utilizadas (Microsoft Internet Explorer utilizando *bubbling* y Netscape utilizando *capture*).

La Figura 4, ilustra de forma gráfica este ciclo de vida y las distintas fases por las que pasa un evento durante su procesamiento.

Cuando un nodo registra varios manejadores para procesar el mismo evento, el orden de ejecución coincide con el orden de registro.

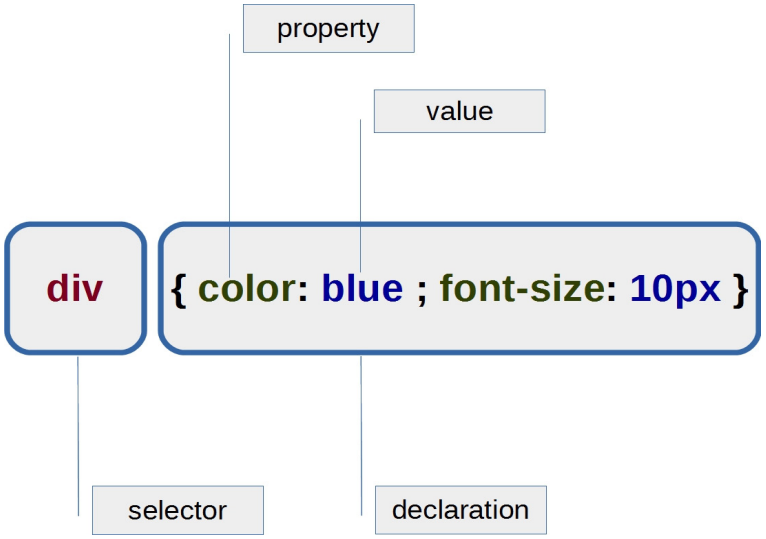


Figura 5. Ejemplo de regla en una hoja de estilo CSS.

Durante la ejecución de un evento, es posible que se generen nuevos eventos que serán procesados de forma síncrona y en orden secuencial, una vez que han finalizado todos los manejadores del evento en curso.

Además de los eventos producidos por acciones de un usuario sobre la página actual, existen otro tipo de eventos que el navegador web genera de forma automática.

Un ejemplo de esta situación es el evento *load*, generado de manera automática por el navegador sobre el elemento *body*, cuando finaliza el proceso de carga de la página. De aquí en adelante, este tipo de eventos se denominarán "eventos automáticos".

2.2.1.2 Hojas de estilo (CSS)

CSS (Cascading Style Sheet en inglés) es un lenguaje para describir el formato y la apariencia de los elementos que forman un documento HTML (por ejemplo, el tamaño de la fuente, el color o el espaciado entre elementos).

El lenguaje CSS permite la separación entre el contenido del documento y su presentación. Una hoja de estilos, contiene una lista de reglas y cada una de estas reglas contiene un selector y un bloque de declaración.

El selector identifica al elemento o grupo de elementos del árbol DOM sobre los cuales aplica la regla y el bloque de declaración contiene una o más propiedades, separadas por un punto y coma.

La Figura 5, ilustra un ejemplo de una regla contenida en una hoja de estilo CSS y en ella, se pueden observar sus componentes principales:

1. Selector: permite identificar al elemento o elementos sobre los cuales se aplicará la regla. Entre otros, los elementos se pueden identificar por:
    - El nombre de la etiqueta HTML.

- Por el atributo *id*. En este caso, el selector comenzará por el caracter #.
- Por el atributo *class*. En este caso el selector comenzará por un punto.

2. Declaración: contiene una o más propiedades separadas por un punto y coma.
3. Propiedad: existen un gran número de propiedades soportadas en los navegadores tradicionales, todas ellas referidas a diferentes aspectos de visualización, como por ejemplo; colores, bordes, fondo de pantalla, textos, animaciones, propiedades de las listas, efectos visuales, paginación, y un largo etcétera.
4. Valor: en algunos casos, los valores se expresan como cadenas de caracteres, en otros casos pueden representar valores numéricos. En el caso de valores numéricos, además del número se indica también la unidad (por ejemplo, *px* representa el número de píxeles).

Entre los atributos visuales que se pueden especificar mediante reglas dentro de una hoja de estilo CSS, se pueden destacar los siguientes:

1. Especificación del área y los bordes de un elemento.
2. Esquema de posicionamiento del elemento dentro del documento (posicionamiento absoluto, relativo, alineamiento, etc.).
3. Efectos visuales a aplicar sobre el elemento (superposiciones, visibilidad, etc.).
4. Generación de contenido, fuentes, tamaños de los textos, etc.

La Figura 6, muestra los diferentes componentes del área que ocupa un elemento del árbol DOM. Las propiedades que se pueden especificar mediante reglas CSS son los márgenes, los bordes y el relleno (“*margin*”, “*border*”, “*padding*”) y se pueden especificar sobre la parte superior (“*top*”), inferior (“*bottom*”), izquierda y derecha. La especificación completa de CSS versión 2 se puede encontrar en [CSS2].

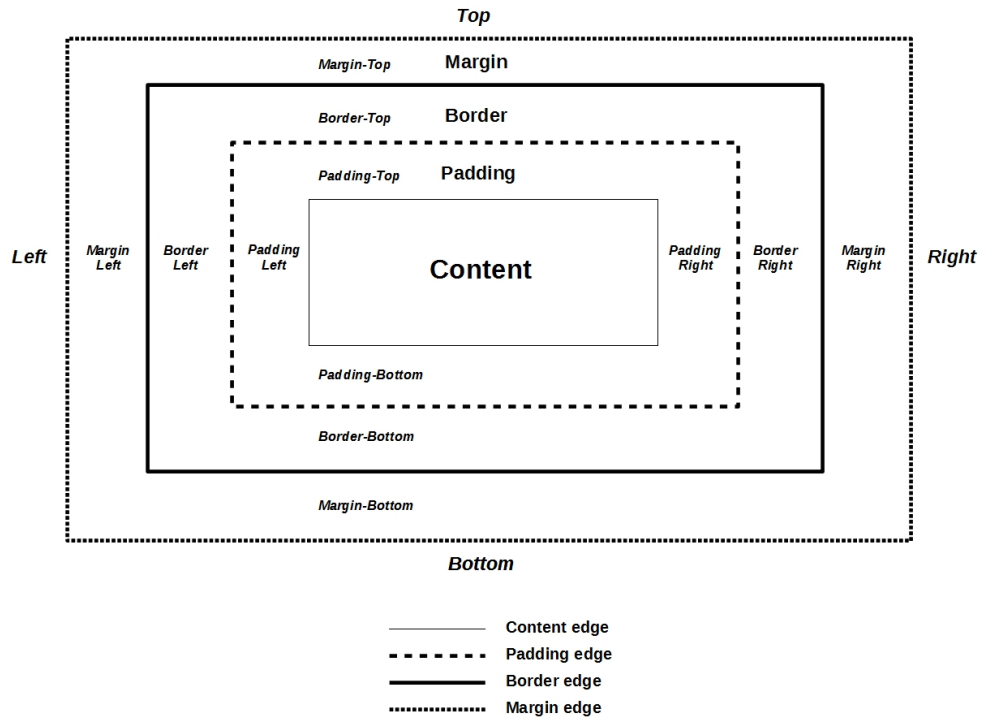


Figura 6. Elementos del área de un nodo que se pueden definir mediante propiedades CSS.

2.2.1.3 Lenguaje JavaScript

JavaScript es un lenguaje de programación interpretado, basado en el estándar ECMAScript [ECMASC].

El uso más importante de este lenguaje es el entorno de los navegadores web, para acceder y manipular el árbol DOM de los documentos HTML, para generar contenidos de manera dinámica, realizar peticiones HTTP asíncronas, etc.

JavaScript soporta estructuras de programación similares a las de lenguajes como C o Java, con la principal diferencia de que el ámbito en el que se define una variable no es el bloque en el que ha sido definida, sino la función en la que ha sido declarada. Este comportamiento puede ser modificado a partir de la versión 1.7 de la especificación.

Un documento HTML, puede incluir código ejecutable en JavaScript utilizando nodos con etiqueta *script*. Estos nodos pueden contener el código ejecutable de forma directa, o pueden hacer referencia a un fichero externo con el código.

Los navegadores web tradicionales, incluyen dentro del contexto de ejecución de JavaScript, una serie de elementos predefinidos:

1. Tipos de datos y objetos predefinidos de uso general.
    a. *Date*: permite manipular fechas.
    b. *Array*: permite trabajar con listas de elementos.

   c. *Math*: permite realizar diversas operaciones matemáticas (máximo, mínimo, etc.)

2. Objetos que permiten manipular el árbol DOM del documento HTML (por ejemplo: el objeto *window* o el objeto *document*).

3. Objetos que permiten realizar otro tipo de operaciones de más alto nivel (por ejemplo, peticiones AJAX).

La Figura 7, muestra los principales objetos accesibles desde el contexto de ejecución de JavaScript de los navegadores convencionales. Entre estos objetos destacan:

1. El objeto *window*: representa la ventana actual y además, también es el contexto de ejecución de más alto nivel.

2. El objeto *document*: representa al documento que está cargado en una ventana. Desde este objeto se puede acceder a todos los elementos del árbol DOM.

Tanto el objeto *document*, como los otros objetos del contexto JavaScript, proporcionan una serie de funciones y métodos que permiten manipular el árbol DOM de la página web.

Por ejemplo, el objeto *document* permite, entre otras muchas funcionalidades, las siguientes operaciones:

1. Acceder al nodo raíz del árbol utilizando la llamada *document.documentElement*.

2. Obtener los nodos de un tipo determinado utilizando la función *getElementsByTagName*.

3. Obtener nodos por su identificador utilizando la función *getElementById*.

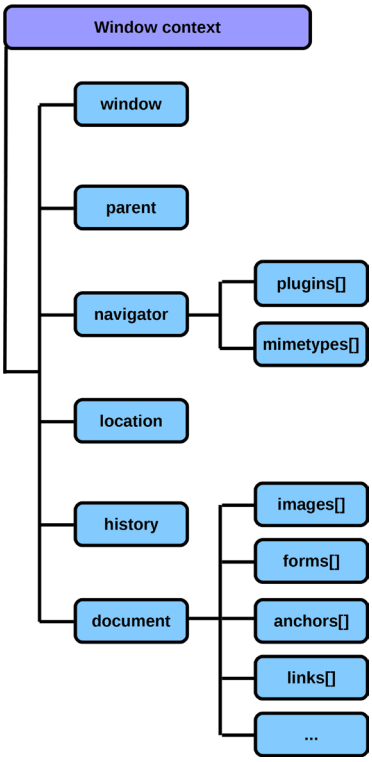


Figura 7. Principales objetos accesibles desde el contexto JavaScript.

2.2.2 Arquitectura de referencia de los navegadores convencionales

Un navegador web, es un programa software desarrollado para la obtención y presentación de contenidos descargados de Internet, sobre todo documentos HTML.

En la actualidad, los navegadores web más populares son: Mozilla Firefox [MOZFF], Microsoft Internet Explorer [MSIE], Google Chrome [GOOCHR], Apple Safari [APPSAF] y Opera [OPERA].

La Figura 8, detalla la arquitectura de referencia de los navegadores web convencionales [GG05] y en ella, se pueden observar sus principales componentes de alto nivel: interfaz de usuario, motor de navegación (también conocido por su acepción en inglés, *browser engine*) y motor de renderización (también conocido por su nombre en inglés, *rendering engine*).

Los componentes de alto nivel, hacen uso de los subsistemas auxiliares que también se muestran en la figura: subsistema de red, subsistema de información persistente, intérprete de JavaScript, procesador de documentos HTML y subsistema de primitivas de visualización (también conocido por su acepción en inglés, *display backen*d).

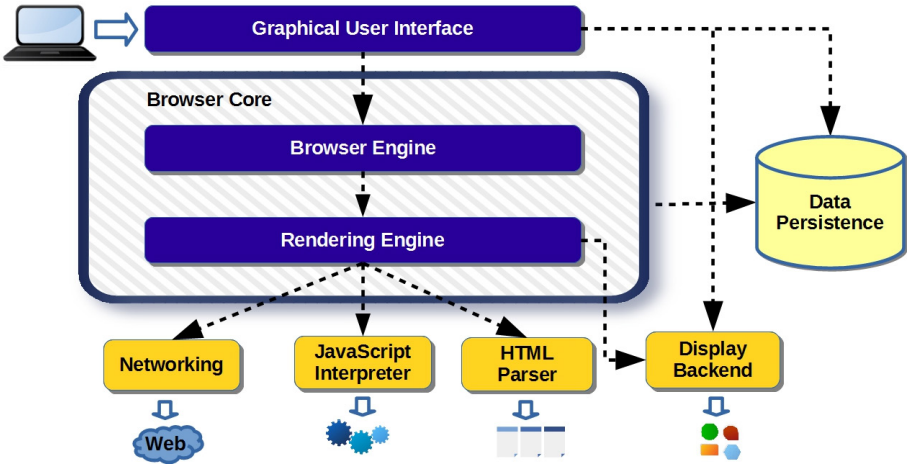


Figura 8. Arquitectura de los navegadores tradicionales.

A continuación, se describen de forma breve, las principales características de todos estos componentes:

1. Interfaz de usuario (*graphical user interface*): incluye el área del navegador excepto la ventana principal en la que se muestra la página HTML que está siendo renderizada. Esto incluye el área de direcciones, las barras de herramientas, el panel con los menús, el panel con los marcadores (*bookmarks*), etc.

2. Motor de navegación (*browser engine*): interfaz de alto nivel para consultar y manipular el motor de renderización.

   Este componente, proporciona un API para invocar acciones de alto nivel del navegador, por ejemplo, iniciar la carga de una URL, ir a la página anterior o recargar la página actual.

   Entre los motores de navegación más populares destacan sobre todo:

   - Blink [GOOBLK] utilizado en Google Chrome y Opera.
   - WebKit [WEBKIT] utilizado en Apple Safari.
   - Gecko [MOZGCK] utilizado en Mozilla Firefox.

3. Motor de renderización (*rendering engine*): es el responsable de procesar y mostrar los contenidos HTML descargados de la Web.

   Los motores de renderización de los navegadores más populares son los siguientes:

   - Gecko [MOZGCK] utilizado en Mozilla Firefox (Gecko engloba motor de renderización y motor de navegación).
   - WebCore utilizado en los navegadores basados en WebKit.
   - Trident [MSTRDT] utilizado en Microsoft Internet Explorer.

4. Subsistema de red (*networking*): proporciona un API de alto nivel que permite ejecutar peticiones HTTP y otras llamadas de red.

5. Intérprete de JavaScript (*JavaScript interpreter*): subsistema utilizado para procesar, compilar y ejecutar los contenidos JavaScript.

   Los motores para el procesamiento de JavaScript más populares que podemos encontrar en los navegadores convencionales son los siguientes:

   - Spider-Monkey [MOZSPD] utilizado en Mozilla Firefox.
   - V8 [V8] utilizado en Google Chrome y Opera.
   - Además, cabe destacar también Mozilla Rhino [MOZRHN], un intérprete de JavaScript desarrollado para el lenguaje de programación Java.

6. Procesador HTML/XML (*HTML parser*): subsistema encargado de procesar todos los documentos HTML y XML descargados de Internet o generados de forma dinámica durante la evaluación del código JavaScript.

7. Subsistema de persistencia (*data persistence*): proporciona un mecanismo para el almacenamiento de toda la información persistente, entre la que podemos destacar:

   - Las cookies.
   - Los objetos almacenados en la caché.
   - Los parámetros de configuración del navegador.

8. Primitivas de visualización (*display backend*): proporciona un API genérico para dibujar, en la ventana del navegador, componentes de bajo nivel (conocidos también por su denominación en inglés, *widgets*).

   Para este subsistema, es habitual la utilización de un API genérico (el mismo para todos los sistemas operativos), junto con una implementación nativa de los objetos más básicos para cada plataforma.

2.2.3 Motor de renderización de los navegadores convencionales

El motor de renderización, representa el núcleo central de los navegadores web al ser el componente encargado de procesar y dibujar los contenidos HTML descargados de Internet.

Durante el procesado de las páginas descargadas, se disparan una serie de eventos en cascada, la mayoría de los cuales son procesados en orden secuencial.

En primer lugar, el contenido referenciado por la URL se descarga y descomprime. A continuación, el diseño de la página (también conocido por su nombre en inglés, *layout*) se calcula de forma dinámica y se va dibujando en la interfaz de usuario del navegador.

Durante el procesado de la página HTML, los diferentes contenidos se van descubriendo de forma incremental (imágenes, scripts, hojas de estilo CSS, etc.).

Durante la ejecución del código JavaScript, se interactúa con el diseño de la página y este diseño puede verse alterado de manera dinámica.

Por la semántica del lenguaje JavaScript, los navegadores ejecutan los scripts contenidos en el documento HTML de manera secuencial, salvo algunos casos especiales que se detallan a continuación:

1. Por un lado, los scripts que contienen el atributo *async*: estos scripts se pueden ejecutar de manera asíncrona, sin esperar a que otros scripts finalicen su ejecución.

   Esta funcionalidad ha sido introducida en el estándar HTML5 [W3CHTML5] y está disponible sólo para scripts referenciados por URL externas (scripts con el atributo *src* definido).

2. Por otro lado, los denominados Web Workers, permiten la ejecución de contenido JavaScript en segundo plano. Estos componentes también forman parte de la especificación del estándar HTML5 y están soportados en los principales navegadores convencionales (incluido Microsoft Internet Explorer desde la versión 10).

   Un Web Worker puede ejecutar código JavaScript en segundo plano, pero con ciertas restricciones. La limitación más importante que presentan estos componentes es que no pueden acceder a los objetos del árbol DOM de una página web.

La Figura 9, expone las diferencias en el contexto de ejecución de un Web Worker, con respecto al contexto de ejecución normal, durante la evaluación de JavaScript cuando el navegador procesa el documento HTML.

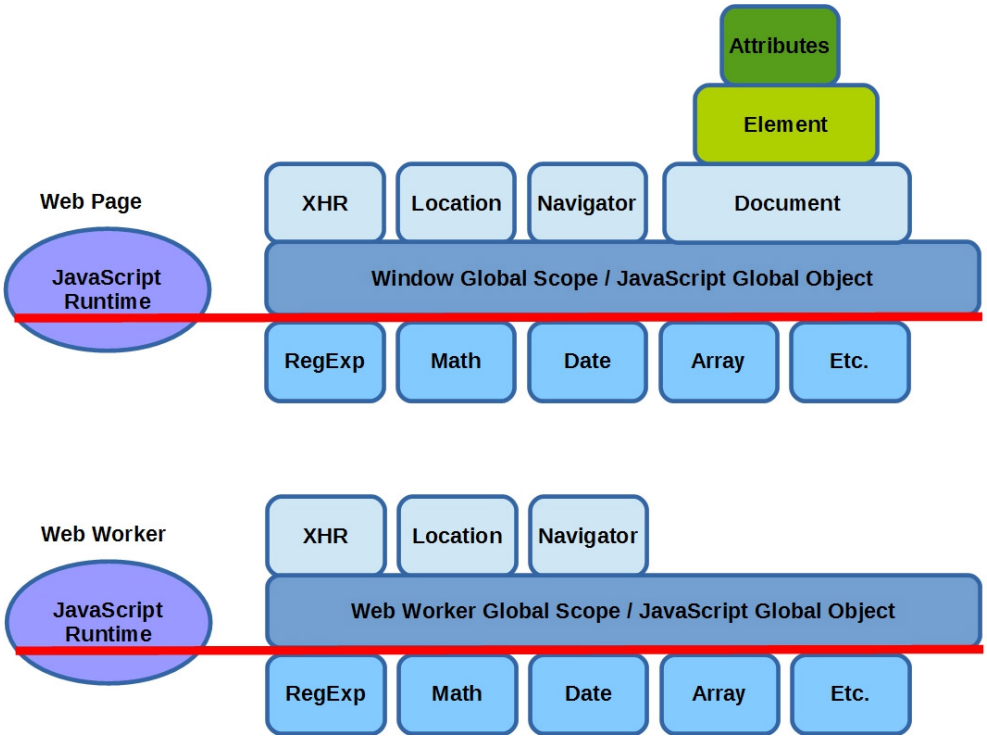


Figura 9. Contexto de ejecución de los Web Workers.

Más en detalle, los Web Workers pueden acceder a los siguientes elementos del contexto de ejecución del motor de JavaScript:

1. Pueden acceder a los objetos predefinidos *navigator* y *location*.
2. Pueden realizar peticiones AJAX, utilizando el objeto predefinido *XMLHttpRequest*.
3. Pueden utilizar distintos tipos de datos y funciones nativas: *Array*, *Date*, *Math* y *String*.
4. Pueden ejecutar JavaScript mediante temporizadores, utilizando las funciones *setTimeout* y *setInterval*.
5. Pueden importar otros scripts utilizando la función predefinida *importScripts*.
6. Pueden ejecutar otros Web Workers.

Y entre las principales limitaciones, se pueden destacar las siguientes:

1. No tienen acceso al árbol DOM de la página.
2. No tienen acceso a los objetos predefinidos *window*, *document* y *parent*.

El motor de renderización de los navegadores convencionales, pasa por diferentes etapas durante el procesamiento de un documento HTML. La Figura 10 muestra estas etapas, que se enumeran y describen a continuación:

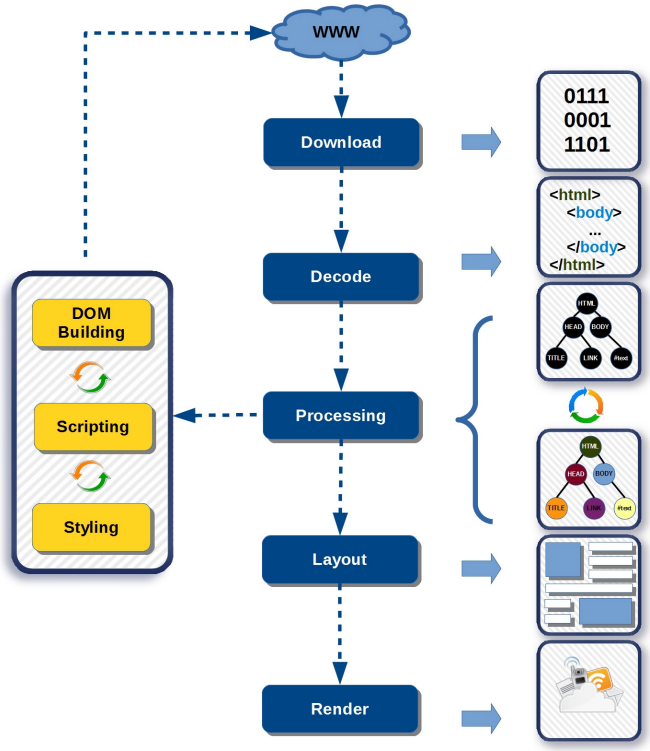


Figura 10. Etapas de procesamiento en el motor de renderización.

1. Descarga y descodificación (*download/decode*): los contenidos HTML se descargan del sitio web.

   Estos contenidos suelen transferirse comprimidos para minimizar la cantidad de datos enviados a través de la red.

2. Procesamiento (*processing*): durante esta etapa se construye el árbol DOM de la página y se procesan scripts, hojas de estilo, etc.

   - Generación del árbol DOM: por motivos de eficiencia, este proceso se ejecuta de manera incremental, en los principales navegadores (por ejemplo, utilizando un procesador SAX [SAXAPI] en lugar de uno de tipo DOM [W3CDOMP]).

     A medida que se van descubriendo nuevos elementos de la página, éstos se descargan y se procesan (por ejemplo: imágenes, hojas de estilo CSS, ficheros JavaScript, etc.).

     Los navegadores, tienen la capacidad de realizar múltiples descargas en paralelo y además, también aplican otras optimizaciones adicionales durante este procesamiento, por ejemplo el procesado especulativo de Mozilla Firefox [MOZSPPAR].

     Además, es muy importante la utilización de cachés para todos estos tipos de documentos, con el objetivo de evitar descargas de elementos que no han sufrido modificaciones, desde la última vez que se accedió a ellos.

- Aplicación de estilos CSS: las hojas de estilo contienen información necesaria para la correcta visualización de los elementos que se muestran en la pantalla. Cada hoja de estilo CSS contiene una serie de reglas, que se aplican sobre los nodos del árbol DOM.

  Existen trabajos recientes que tratan de optimizar el proceso de aplicación de las reglas CSS, por ejemplo, el paralelismo en la aplicación de reglas CSS [BHNV10].

- Ejecución de scripts: salvo las dos excepciones comentadas con anterioridad, los navegadores ejecutan el contenido JavaScript de manera secuencial, aunque aplicando diversas técnicas de optimización.

  Por ejemplo, la técnica de *script streaming* [GOOSS] de Google Chrome permite paralelizar la compilación de un script con la descarga del fichero de la red (mediante la utilización de un hilo auxiliar).

3. Generación del esquema de diseño (*layout*) y renderización (*render*): el diseño de la página, contiene rectángulos con atributos visuales como por ejemplo, dimensiones y colores.

   El proceso de renderizado, dibuja este diseño en la pantalla del navegador, utilizando las primitivas de visualización de bajo nivel.

   Estas primitivas de visualización, suelen estar disponibles como un API de alto nivel genérico, junto con una implementación específica para cada plataforma.

A continuación, la Figura 11 ilustra un ejemplo real del navegador Google Chrome cargando una página HTML. En esta figura, se pueden observar los eventos que se disparan en el motor de renderización del navegador, durante el proceso de construcción del documento HTML.

Estos eventos, se muestran en las herramientas de desarrollador que proporciona Google Chrome, para acceder a la información de más bajo nivel.

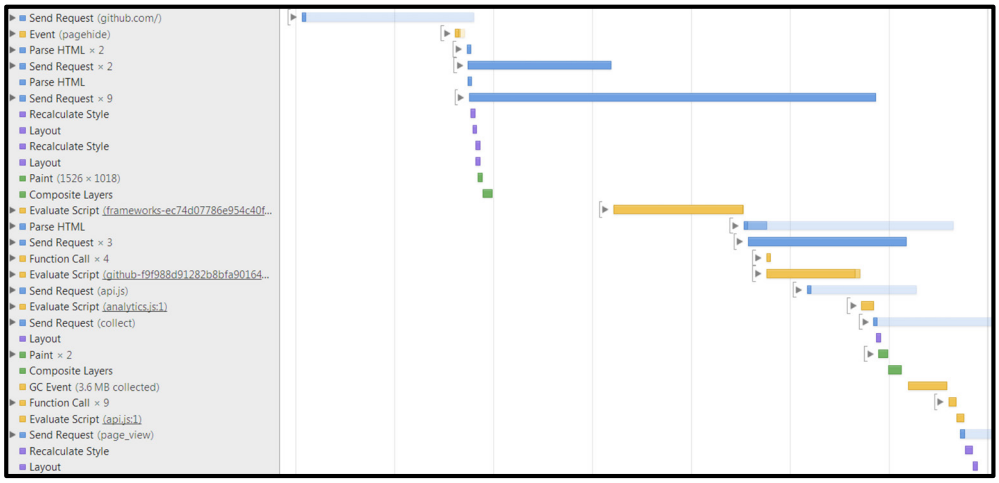


Figura 11. Eventos en Google Chrome cargando una página web.

En la figura, se muestran los eventos de carga ("*Send Request*" y "*Parse HTML*"), entre los que se incluyen eventos de descarga de objetos externos o eventos de procesado del flujo de datos HTML.

Otro tipo de eventos relevantes que aparecen en la figura son los eventos de ejecución de código JavaScript ("*Evaluate Script*" y "*Function Call*").

Y en último lugar, también se pueden observar los eventos de formateado del esquema de diseño ("*Layout*" y "*Recalculate Style*") y los eventos de dibujado en la ventana del navegador ("*Paint*" y "*Composite Layers*").

Como se puede observar en este ejemplo, la página web contiene varias hojas de estilo CSS en la cabecera, así como imágenes y 3 scripts. Esos elementos comienzan a descargarse al principio del proceso (eventos "*Send Request x 2"* y "*Send Request x 9"*) y estas descargas de elementos externos se realizan en paralelo.

Los eventos de procesado de HTML (eventos de tipo "*Parse HTML"* en la figura), representan el tiempo que emplea el navegador en el procesado incremental del flujo de datos HTML. Durante este procesado, se van descubriendo nuevos recursos y disparando nuevos eventos.

Los eventos "*Function Call x 4"* y "*Function Call x 9"*, representan la ejecución de funciones JavaScript, generadas como consecuencia de eventos asíncronos, disparados utilizando las funciones *SetTimeout*/*SetInterval* o como respuesta a eventos generados en la página, durante la ejecución del fichero JavaScript *framework.js*.

La ejecución del código JavaScript se ejecuta, sobre todo, de manera secuencial, aunque existe una pequeña paralelización porque algunos scripts contienen el atributo *async* ("*Function Call x 4"*, se ejecuta en paralelo con la evaluación del script *github.js*).

En la figura, también aparece un cuarto script llamado *api.js*. Este script se genera de forma dinámica, durante la evaluación del script *github.js* (utilizando la función *write* del objeto predefinido *document*). Este nuevo script contiene el atributo

async, con lo que también podría ser evaluado en paralelo una vez que haya sido descargado.

Los eventos de renderización y de dibujado en la ventana del navegador, se generan durante todo el proceso de carga de la página.

El esquema de diseño, se recalcula cada vez que se añaden nuevos nodos al árbol DOM de la página, tanto en la fase de procesado del HTML, como en la fase evaluación del JavaScript.

Cuando se procesan las hojas de estilo y se aplican las reglas CSS, el esquema de diseño también se recalcula.

2.2.4 Arquitectura de los principales navegadores web convencionales

En el siguiente apartado, se analiza con mayor profundidad la arquitectura de los principales navegadores convencionales.

En primer lugar, se analizará el navegador Mozilla Firefox y su motor de renderización Gecko.

A continuación, se analizará el navegador Google Chrome y su motor de renderización Blink (desarrollado a partir del motor de renderización WebKit).

A continuación, se analizará Microsoft Internet Explorer y su motor de renderización Trident.

Y para finalizar, el último navegador analizado será Apple Safari y su motor de renderización de código abierto WebKit (cabe destacar que este motor de renderización también se utiliza en otros muchos navegadores de libre distribución).

2.2.4.1 Arquitectura del navegador Mozilla Firefox

Mozilla Firefox, es un navegador de código abierto desarrollado por la Fundación Mozilla.

Este navegador, funciona en los principales sistemas operativos (Microsoft Windows, Linux, Mac OS X y Android, entre otros).

La Figura 12, detalla la arquitectura completa de Mozilla Firefox.

En la figura, se puede observar como el esquema de sus componentes es muy similar a la arquitectura de referencia descrita en la Figura 8.

Para la renderización de las páginas HTML, Mozilla Firefox utiliza el motor de renderización Gecko, desarrollado en el lenguaje C++ y creado en primera instancia, por la compañía Netscape.

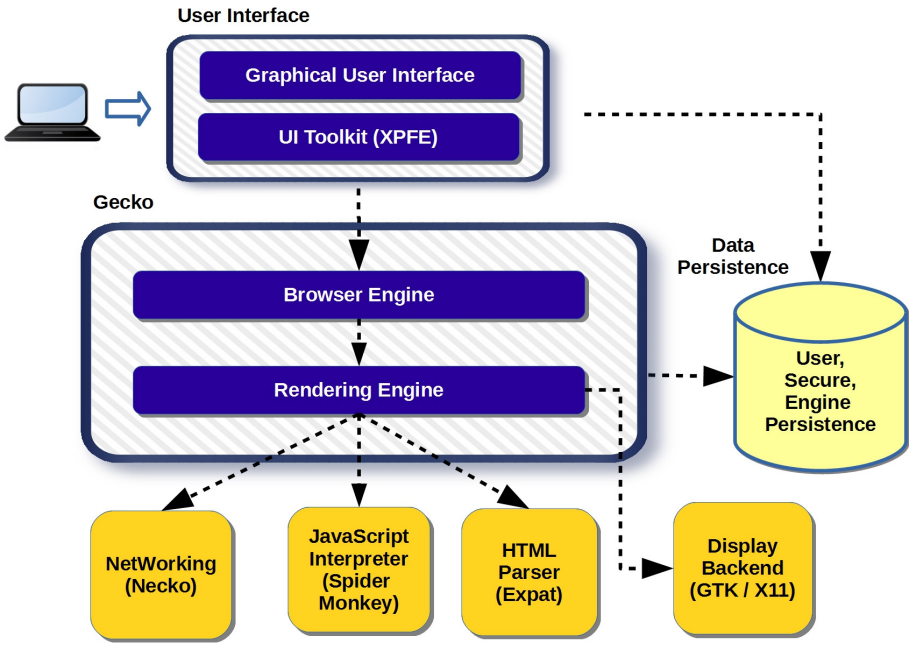


Figura 12. Arquitectura del navegador web Firefox.

Gecko da soporte a una gran variedad de estándares, entre los que destacan HTML 4.01 [W3CHTML4], CSS versiones 1 [CSS1] y 2 [CSS2], DOM nivel 0, 1 y 2 [DOM], XML 1.0 [W3CXML], RDF [W3CRDF], JavaScript 1.5 (basado en el estándar ECMAScript [ECMASC]), HTTP 1.1 [HTTP11] y FTP [FTP], y un largo etcétera.

Gecko es un motor de renderización que utiliza un proceso principal (en este caso se trata de un hilo, no un proceso del sistema operativo) para representar el contenido HTML en la ventana del navegador, aunque algunas tareas se ejecutan de forma íntegra, en hilos auxiliares.

Algunos de los subsistemas principales de este componente son los siguientes:

1. Procesador de documentos HTML y XML (Expat).
2. Subsistema CSS para el procesamiento de hojas de estilo.
3. Motor de JavaScript SpiderMonkey.
4. Librería para el procesamiento de imágenes y de fuentes.
5. Librería de red Necko.
6. Librería para almacenar y acceder a las preferencias de usuario.
7. Librerías específicas con componentes nativos para los distintos sistemas operativos.
8. Subsistema para el desarrollo de complementos (también conocidos como *plugins*) (NPAPI).

En la Figura 13, se muestra el funcionamiento básico de Gecko, una vez que el contenido HTML ha sido descargado utilizando el subsistema de red.

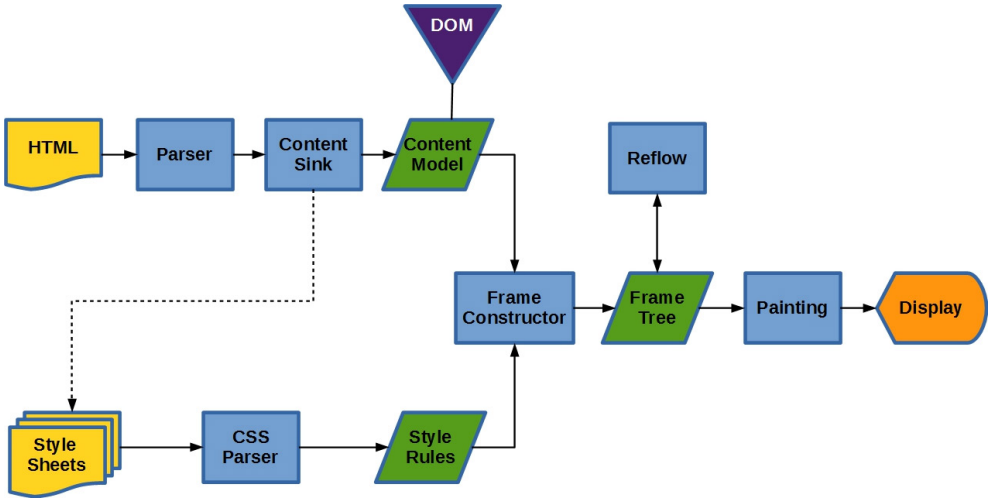


Figura 13. Funcionamiento básico de Gecko.

En ese momento, se inicia el flujo de procesamiento por los diferentes subsistemas hasta que, en último lugar, se muestra el contenido HTML formateado en la ventana del navegador.

En primer lugar, el flujo de datos HTML se envía a los subsistemas "*Parser*" y "*Content Sink*", donde se construye una estructura con el árbol DOM del documento ("*Content Model*") y se detectan otro tipo de elementos como hojas de estilo CSS y scripts.

A medida que son detectadas y descargadas, las hojas de estilo CSS se envían al procesador de CSS, componente encargado de generar una estructura con las reglas extraídas ("*Style Rules*").

A continuación, el componente "*Frame Constructor*" recibe, tanto el árbol DOM, como las reglas de estilo CSS y genera una estructura adicional con la información de presentación de los diferentes nodos que forman parte del árbol DOM de la página.

Cada nodo del árbol DOM, contendrá uno o varios "*frames*" con información de presentación. Cada uno de estos elementos, representa información de un rectángulo (contendrá información del alto, del ancho y de la distancia al marco padre que lo contiene).

El proceso de "*Reflow*", es el encargado de calcular la geometría de todos los elementos que forman parte del árbol de presentación (tamaño y posición).

Estos procesos de "*Reflow*", comienzan en la raíz del árbol de presentación y se propagan de manera recursiva, hasta los nodos hoja.

Este proceso también se puede desencadenar con posterioridad, como consecuencia de acciones de usuario o de acciones desencadenadas durante la evaluación de JavaScript (y puede desencadenarse, tanto en la raíz, como en un elemento más interno).

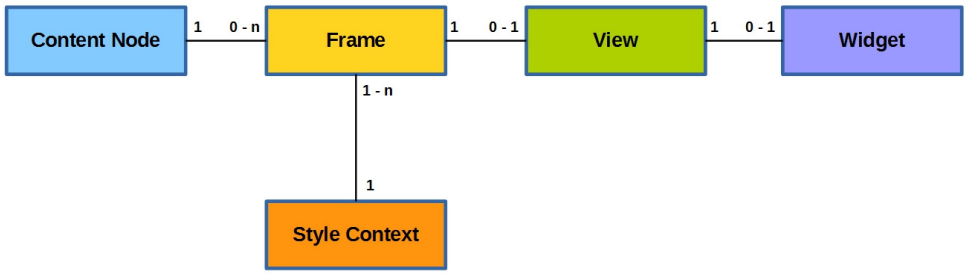


Figura 14. Estructura de objetos de Gecko.

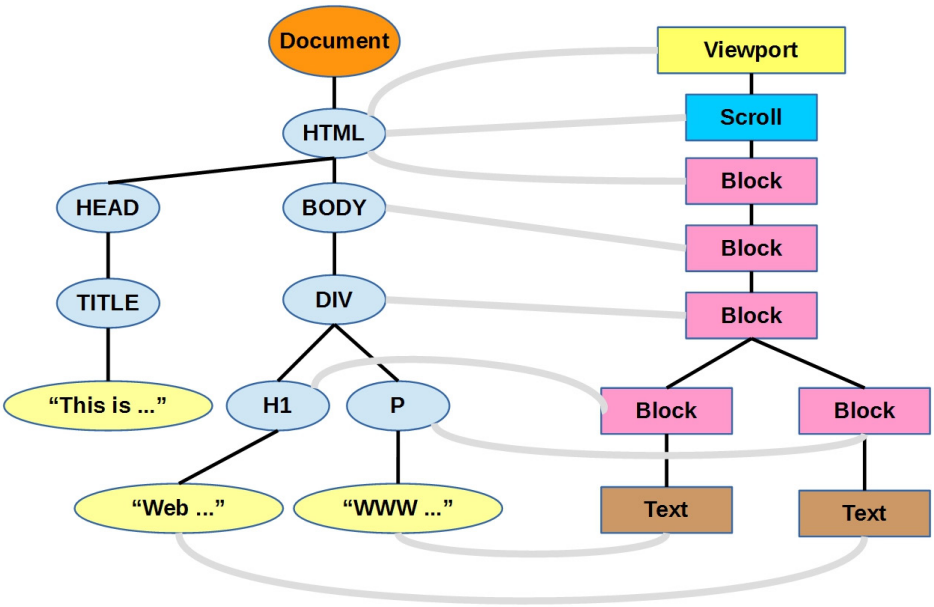


Figura 15. Ejemplo de árbol de visualización en Gecko.

La Figura 14, detalla la estructura de objetos que se genera en Gecko para la renderización de la página HTML.

Cada uno de los nodos del árbol DOM, puede contener uno o varios marcos ("*Frame*"), de forma rectangular y con información geométrica. El contexto de estilo ("*Style Context*") contendrá información no geométrica.

Además, cada uno de los marcos podrá incluir propiedades de visualización ("*View*"), con información adicional de presentación (por ejemplo: transparencias, z-order, etc.).

En último lugar, cada una de las vistas podrá contener una ventana nativa ("*widget*"), para su visualización en la pantalla del navegador.

La Figura 15, muestra un ejemplo de árbol de renderización generado a partir de un árbol DOM de una sencilla página HTML de ejemplo.

En ese árbol de renderización, se pueden observar distintos tipos de objetos de renderización y además, también se observa como algunos de los nodos del árbol

DOM, generan más de un objeto dentro del árbol de renderización. El marco contenedor de más alto nivel se denomina "*Viewport*".

Aunque Gecko es un motor de renderización con un *thread* principal (mono-hilo), algunas tareas se ejecutan de forma íntegra en *threads* auxiliares.

Por ejemplo, el procesado del código fuente HTML, se realiza en un hilo dedicado en exclusividad a esta tarea. Sin embargo, los contenidos HTML generados de forma dinámica desde JavaScript, utilizando la función predefinida *write*, se procesan de manera síncrona en el *thread* principal.

El procesador de HTML de Gecko, implementa optimizaciones adicionales, como por ejemplo la carga especulativa de scripts, hojas de estilo, imágenes, vídeos, etc. [MOZSPPAR].

Esta optimización se utiliza en aquellos casos en los que el procesador de HTML se ejecuta en un *thread* dedicado, donde a medida que va descubriendo elementos pre-cargables, se van generando acciones de pre-carga, que se almacenarán en una cola específica. Una vez generadas, estas acciones, se ejecutan en paralelo con el procesado del contenido HTML.

La versión actual de Firefox, utiliza el mismo proceso del sistema operativo para todas las pestañas abiertas y también utiliza este mismo proceso para las tareas de renderización de la interfaz de usuario (la ventana del navegador con los menús, las barras de herramientas, etc.).

Las versiones de desarrollo, todavía inestables, incluyen un prototipo que permite utilizar procesos diferentes en la interfaz de usuario y en cada una de las pestañas abiertas (cada pestaña muestra un documento HTML). Este proyecto se denomina Electrolysis Firefox [ELEFF].

2.2.4.2 Arquitectura del navegador Google Chrome

Google Chrome, es un navegador desarrollado por Google. Este navegador hace uso de varios componentes de código abierto, como por ejemplo, su motor de renderización Blink o el intérprete de JavaScript V8.

Google Chrome, también soporta los principales sistemas operativos, tanto de escritorio como de plataformas móviles.

En sus primeras versiones, Google Chrome integraba el motor de navegación de código abierto WebKit, pero con posterioridad, ha creado una versión a partir de éste, a la que ha denominado Blink.

En la Figura 16, se puede observar cómo éste utiliza distintos procesos del sistema operativo para cada una de las pestañas abiertas, así como un proceso adicional que actúa como coordinador.

Este proceso controlador, se comunica con cada uno de los procesos hijos, utilizando un mecanismo de comunicación basado en el envío de mensajes ("*IPC*" en la figura) y mantiene un objeto "*RenderProcessHost*" por cada uno de estos hijos.

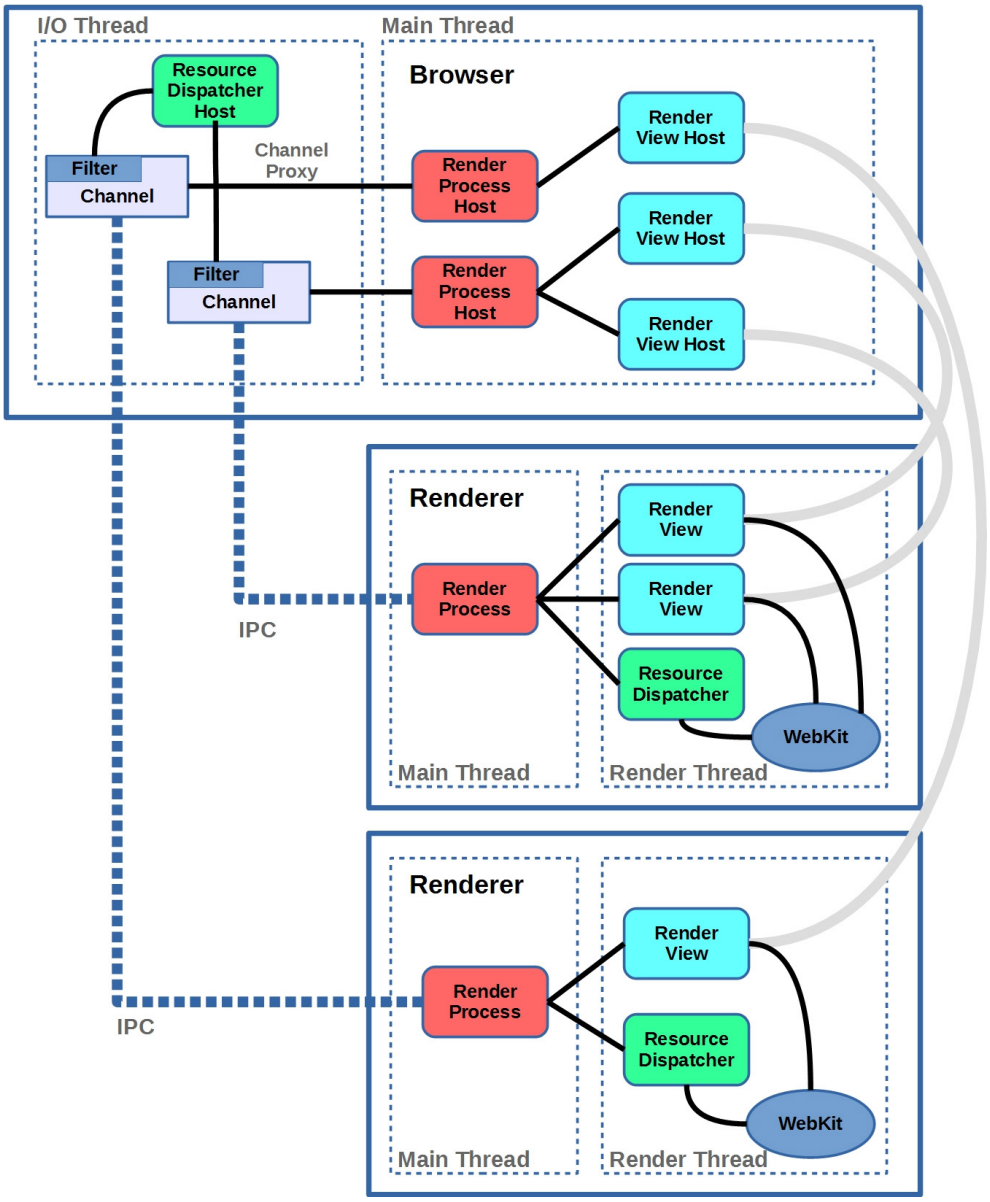


Figura 16. Arquitectura del navegador Google Chrome.

El motor de navegación WebKit/Blink se ejecuta en el hilo "*Render thread*" mientras que en el hilo "*Main thread*" se mantiene un objeto "*Render Process*" para la comunicación con el proceso controlador.

La Figura 17, ilustra la pila de procesos de Google Chrome y WebKit/Blink.

En esta figura, se observa como toda la parte de renderización de WebKit reside en el mismo proceso del sistema operativo.

En el apartado 2.2.4.4, se muestran las diferencias con otras implementaciones del mismo motor de renderización en otros navegadores también basados en WebKit.

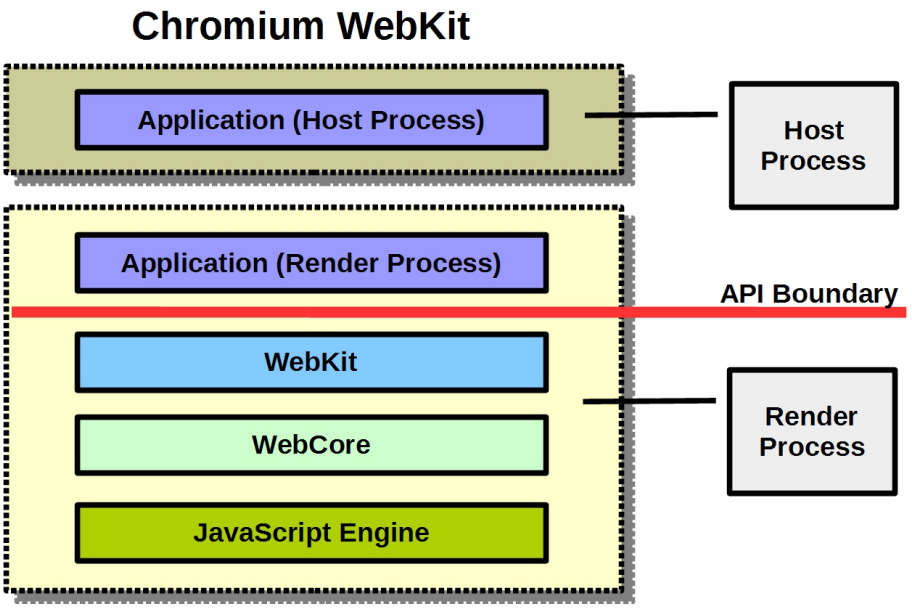


Figura 17. Arquitectura de WebKit en Google Chrome.

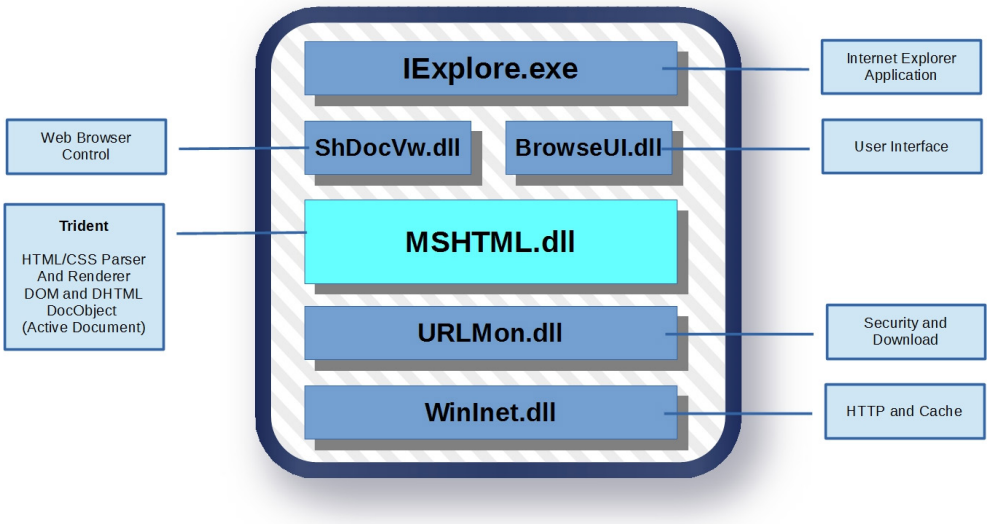


Figura 18. Arquitectura del navegador Internet Explorer.

2.2.4.3 Arquitectura del navegador Microsoft Internet Explorer

Internet Explorer, es un navegador web desarrollado por la empresa Microsoft y que tiene soporte, casi de manera exclusiva, en los sistemas operativos de esta compañía.

Este navegador, en sus últimas versiones, también utiliza diferentes procesos del sistema operativo para cada una de las pestañas (además de un proceso controlador para la gestión de todas ellas).

La arquitectura general de cada uno de estos procesos se muestra en la Figura 18. En ella, se pueden ver los elementos principales del navegador:

1. IExplore.exe: *thread* ejecutable de más alto nivel, que delega en los otros componentes de bajo nivel, los trabajos de renderización, navegación, etc.
2. BrowserUI.dll: librería que proporciona la interfaz de usuario del navegador (menús, barras de herramientas, etc.).
3. ShDocVw.dll: librería que proporciona funcionalidades de alto nivel, como por ejemplo, navegación y acceso al historial.

   Esta librería, ofrece también distintos tipos de interfaces para acceder al navegador desde aplicaciones desarrolladas en lenguajes propietarios de Microsoft (por ejemplo, lenguajes .NET).
4. Mshtml.dll (Trident): librería con el motor de renderización del navegador y pieza fundamental dentro de la arquitectura de Microsoft Internet Explorer.

   En sus últimas versiones, proporciona un amplio soporte a estándares como CSS versión 3 [CSS3], HTML versión 5 [W3CHTML5], XHTML [W3CXHTML] o SVG [W3CSVG] o incluso WebGL [WEBGL] o SPDY [SPDY].
5. UrlMon.dll: librería para descargar y manejar los diferentes tipos de contenido MIME soportados.
6. WinInet.dll: librería que ofrece primitivas para ejecutar peticiones HTTP y FTP, así como para gestionar la caché del navegador.

El motor de JavaScript que incluye Internet Explorer en sus últimas versiones, se denomina Chakra y con él, Microsoft trata de mejorar la eficiencia y equipararse a sus competidores.

2.2.4.4 Arquitectura del navegador Apple Safari y otros navegadores basados en WebKit

Safari es un navegador desarrollado por Apple y en la actualidad, tiene soporte en los sistemas operativos OS X e iOS.

Safari utiliza WebKit como motor de navegación. Este componente, ha ido evolucionando desde una arquitectura de un solo proceso, a una arquitectura en la que se utilizan varios procesos del sistema operativo, con el objetivo principal de mejorar la estabilidad del sistema.

La Figura 19, muestra la arquitectura tradicional de Safari en la que sólo existe un único proceso del sistema operativo para controlar toda la aplicación. En la figura, se pueden observar los componentes de alto nivel del navegador.

WebKit ofrece un API de alto nivel disponible para el lenguaje de programación C, que permite controlar el componente WebCore que es el que se encarga, en último término, del proceso de renderizado de los documentos HTML.

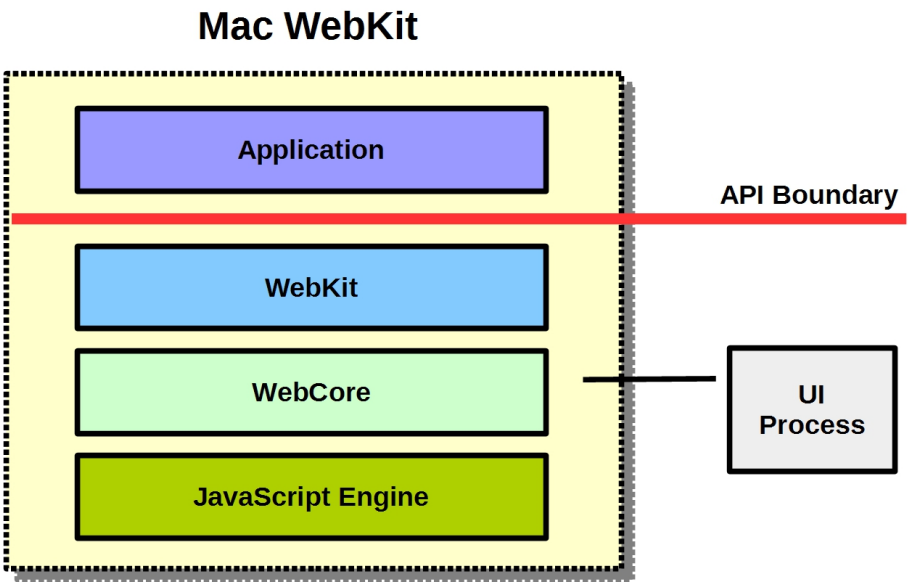


Figura 19. Arquitectura tradicional de WebKit en Safari.

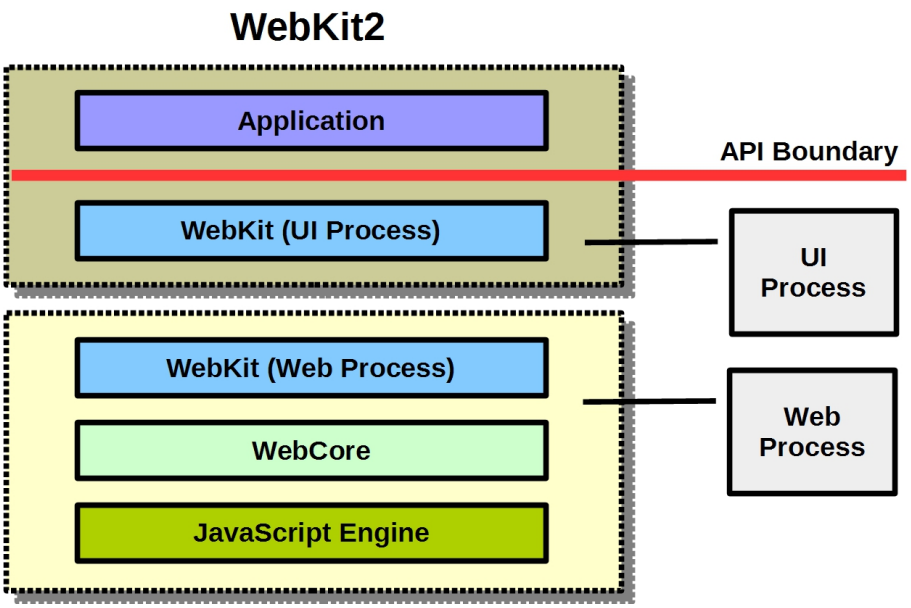


Figura 20. Arquitectura de dos procesos de WebKit2.

La versión 2 de WebKit, proporciona un API nuevo, basado en la utilización de dos procesos diferentes del sistema operativo.

En uno de estos procesos, reside WebCore y toda la parte encargada de la renderización del contenido web y en el otro proceso, reside la interfaz de usuario de la aplicación.

Este nuevo diseño de WebKit2, se ilustra en la Figura 20. En esta figura, se observa como una parte de WebKit2 reside en el proceso de la interfaz de usuario y la otra reside en el proceso de renderización junto con WebCore y el motor de JavaScript.

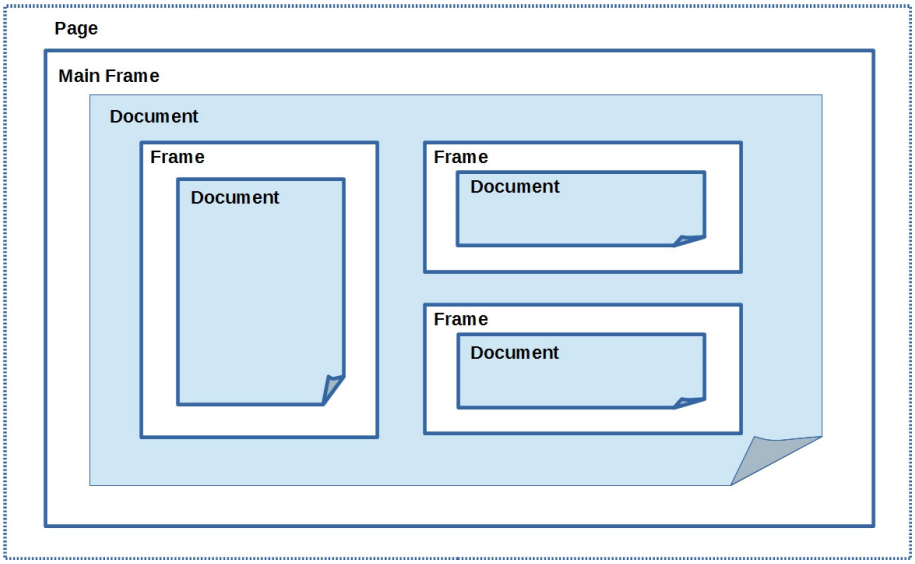


Figura 21. Estructura de páginas y marcos en WebCore.

En cuanto al funcionamiento de WebCore, éste está basado en un modelo de dos hilos o *pipelines*, uno destinado a la carga del documento principal y otro destinado a la carga de los diferentes sub-recursos (como por ejemplo, imágenes y scripts).

Cada página web renderizada, contendrá un marco principal (*frame*) con el documento HTML y un marco adicional por cada subdocumento contenido en este documento principal.

Cada uno de los marcos, pasa por una serie de estados desde el estado inicial de no-inicializado hasta el estado final de cargado.

WebCore también utiliza un mecanismo de precarga para detectar, lo antes posible, los elementos externos contenidos en un documento HTML, e iniciar así, el procesamiento en las etapas iniciales de la construcción del árbol DOM.

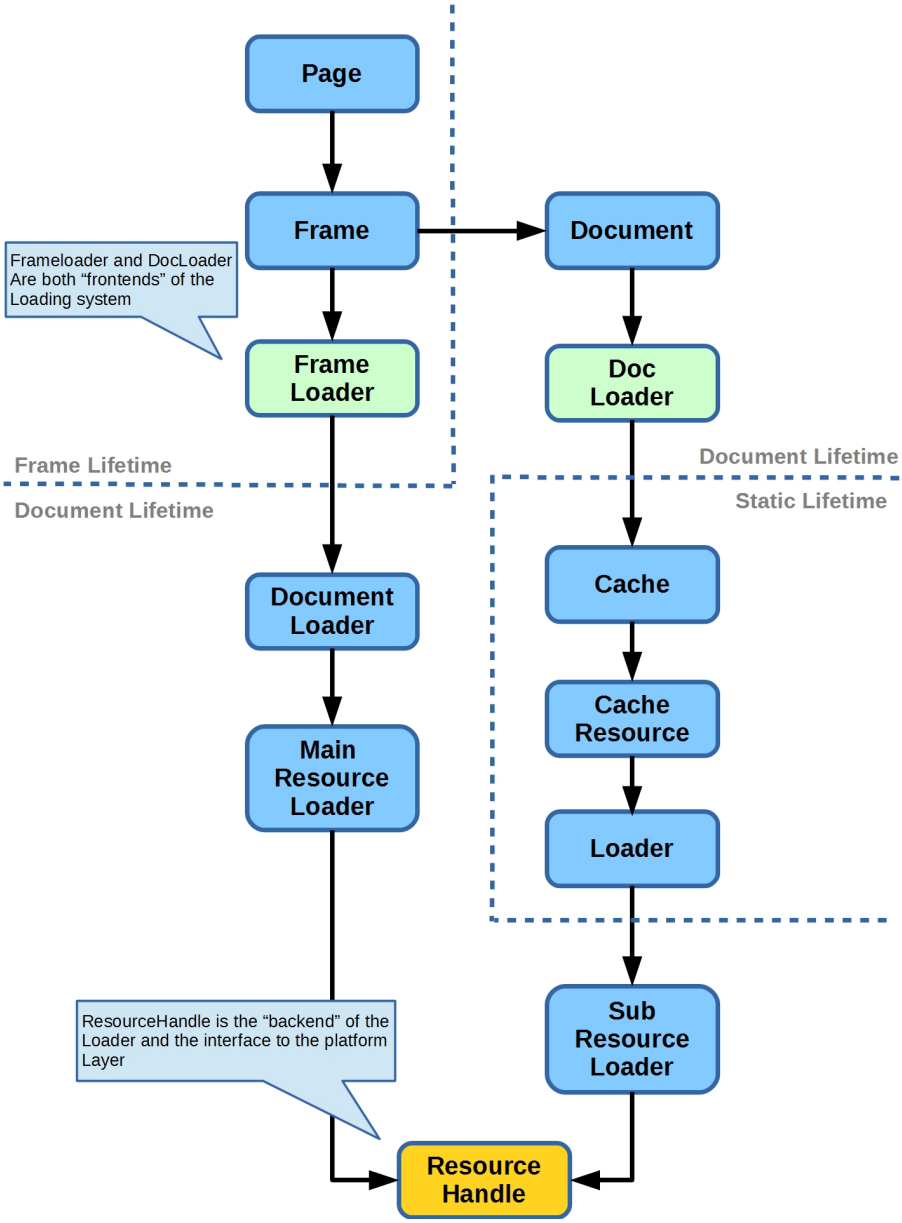


Figura 22. Arquitectura de WebCore.

La Figura 22, detalla el esquema de funcionamiento de WebCore basado en la utilización de dos *pipelines*, uno de ellos para la carga del documento principal (*MainResourceLoader*) y otro, utilizado para la carga de los diferentes sub-recursos (*SubResourceLoader*).

Sobre el proceso de renderización del árbol DOM de la página web, WebCore construirá un árbol adicional, formado por bloques de renderización o *render blocks*. Este árbol, se construye aplicando las reglas CSS sobre los diferentes nodos del árbol DOM.

La Figura 23, muestra un ejemplo de árbol de renderización en WebCore.

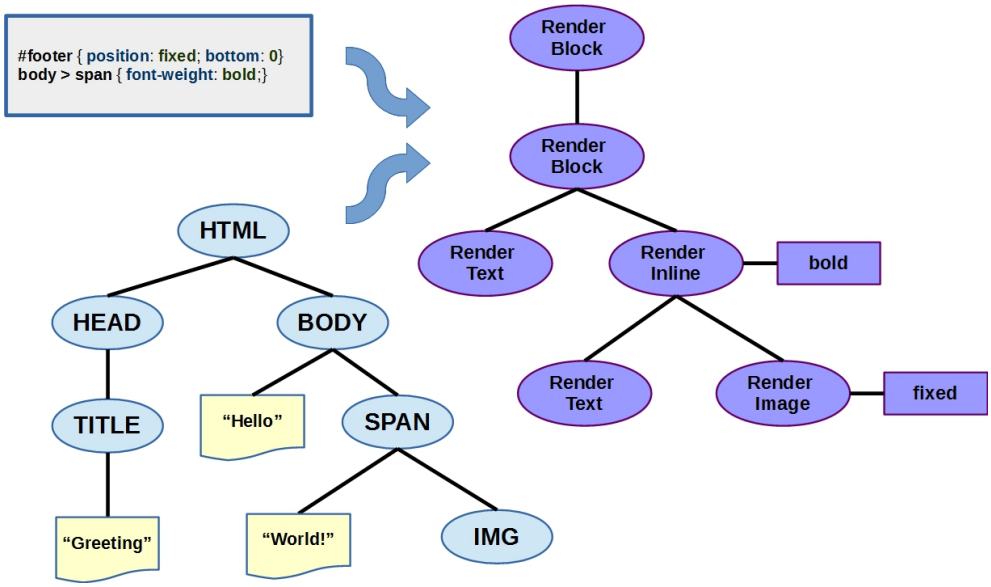


Figura 23. Árbol de renderización en WebCore.

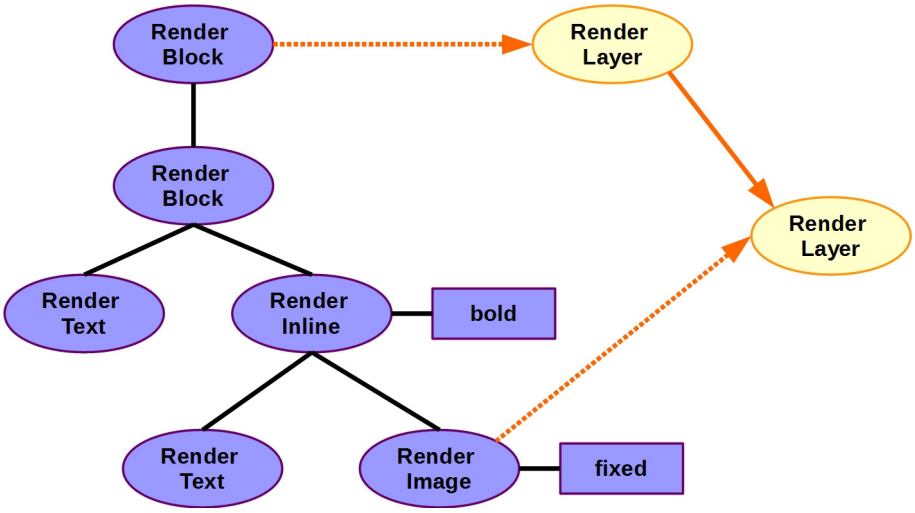


Figura 24. Capas de renderización en WebCore.

De manera similar a Gecko, el árbol de renderización no tiene una correspondencia directa con el árbol DOM, y puede que, algunos nodos del DOM no tengan correspondencia directa en el árbol de renderización.

Por último, para mostrar el contenido en la ventana del navegador, WebCore construye una estructura de capas de renderización (*render layers*) a partir del árbol de renderización.

Esta estructura, permite renderizar cada una de las capas por separado y realizar, con posterioridad, una composición de todas ellas. La Figura 24, ilustra un ejemplo típico de generación de capas, a partir de un árbol de renderización.

Cabe destacar que, además de Apple Safari, existe una gran variedad de navegadores de código abierto basados en el motor de renderización WebKit.

Entre estos navegadores, es posible destacar:

1. PhantomJS: navegador sin interfaz de usuario, muy utilizado en tareas de automatización web, sobre todo para pruebas automatizadas de aplicaciones web.
2. Konqueror: navegador incluido en algunas distribuciones de Linux.
3. El navegador que se incluye en los dispositivos Kindle de Amazon.

2.3 AUTOMATIZACIÓN WEB BASADA EN NAVEGADORES TRADICIONALES

A continuación, se detallan los principales sistemas de automatización web existentes a día de hoy, que basan su funcionamiento en la utilización del API de alto nivel de alguno de los navegadores convencionales del apartado 2.2.

2.3.1 iMacros

iMacros [IMACROS] es una herramienta de automatización web, que proporciona una extensión para los navegadores Mozilla Firefox, Google Chrome e Internet Explorer. Además, también proporciona un componente para incrustar dentro de aplicaciones desarrolladas en alguno de los lenguajes de programación .NET.

Este software, permite generar y reproducir con posterioridad secuencias de navegación, también denominadas “macros”.

La Figura 25, muestra una captura de pantalla de iMacros reproduciendo una secuencia de navegación web, a través del componente proporcionado para el navegador Mozilla Firefox.

En esta figura, se puede observar el panel con la secuencia que se está generando (en la parte izquierda) y la sintaxis específica que utiliza iMacros a la hora de capturar los eventos del usuario y grabar las acciones.

En este caso, se trata de una secuencia que, en primer lugar, accede a la página de inicio del sitio web de la Wikipedia. A continuación, selecciona la versión en inglés y accede a la página de búsqueda, y en último lugar, rellena de forma automática el formulario de búsqueda con el término “*World Wide Web*”.

Una vez que el navegador se ha situado en la página con la lista de resultados, la secuencia de navegación incluye un comando para seleccionar el primero de ellos.

En iMacros, la secuencia de navegación se puede almacenar y modificar, para reproducir más adelante de forma automática.

Esta herramienta, también permite extraer información de las páginas a las que se accede, generar informes con el resultado de la ejecución de la secuencia, etc.

De forma adicional, la versión Enterprise de iMacros, instala un API para controlar el navegador web desde cualquier lenguaje de programación de Microsoft Windows con soporte para objetos COM.

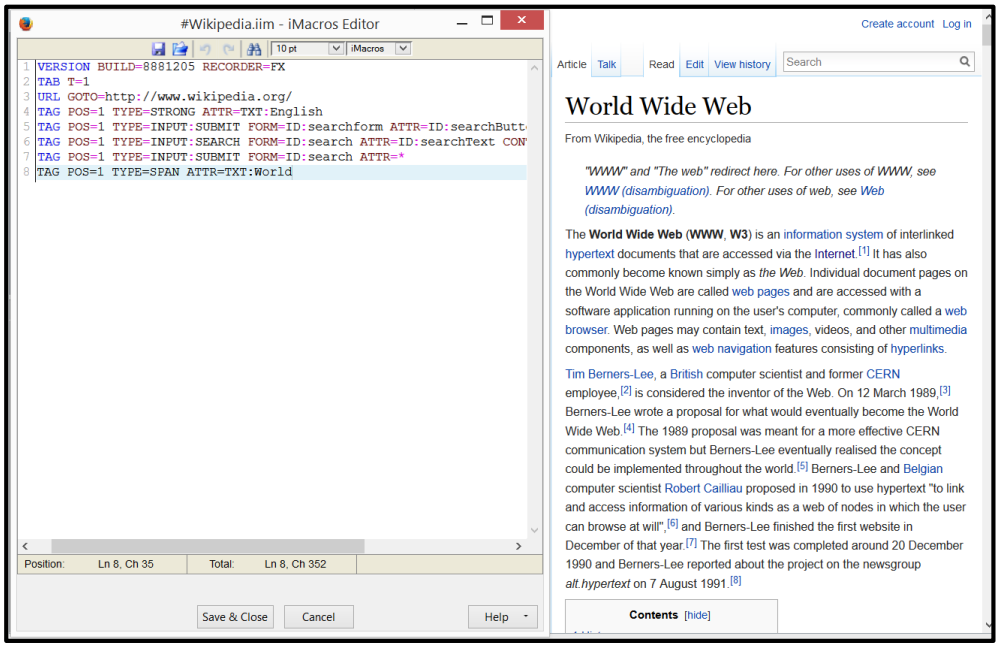


Figura 25. Ejemplo de secuencia de navegación web en IMacros.

2.3.2 Kapow

Kapow [KAPOW] es un conjunto de herramientas, desarrollada por la empresa Kapow Technologies, enfocadas en la creación de aplicaciones de integración web.

Este conjunto de herramientas, conforman la plataforma Kapow (*Kapow Platform*), y en ella, se proporcionan distintas soluciones, entre las que destaca la herramienta de automatización web denominada Kapow Katalist.

Esta herramienta Kapow Katalist, está compuesta por diferentes módulos que se detallan a continuación:

1. Design Studio: herramienta gráfica que permite crear procesos de automatización web (denominados “robots”).
2. RoboServer: servidor de ejecución que permite la ejecución de las tareas de automatización web creadas con la herramienta Design Studio.
3. Extraction Browser: componente que permite integrar la información extraída tras la ejecución de los “robots” con las aplicaciones que consumirán esa información.
4. Management Console: herramienta de administración que permite planificar la ejecución de las tareas, monitorizar el rendimiento del sistema, etc.

Este sistema de automatización web, proporciona dos componentes de navegación diferentes, un navegador clásico (basado en un navegador desarrollado a medida), orientado a las aplicaciones web antiguas y un navegador avanzado (basado en WebKit) orientado a las aplicaciones web más modernas.

2.3.3 Mozenda

Mozenda [MOZENDA] es una herramienta comercial que permite automatizar tareas de extracción sobre fuentes web.

Mozenda proporciona una aplicación de escritorio denominada *Web Agent Builder*, que permite construir las tareas de automatización web (también llamados agentes web) y ofrece además, una aplicación web denominada *Mozenda Web Console*, que permite acceder a los datos extraídos.

Para la generación de los agentes web, Mozenda proporciona un navegador (Microsoft Internet Explorer) incrustado dentro de la aplicación *Web Agent Builder*.

Una vez construidos los agentes, éstos se pueden ejecutar de forma manual o puede planificarse su ejecución de manera automática.

2.3.4 QEngine

QEngine [QEGN] es una herramienta comercial para la automatización de pruebas funcionales y para medir el rendimiento de aplicaciones y servicios web.

En QEngine, una secuencia de navegación (denominada "script") es un programa codificado en un lenguaje de scripting propietario.

La herramienta *QEngine Web Functional Test*, incluye una barra de herramientas que se puede instalar en Internet Explorer o en Mozilla Firefox.

En la actualidad, esta herramienta se encuentra en fase de fin de ciclo de vida.

2.3.5 Sahi

Sahi [SAHI] es una herramienta de código abierto, desarrollada para la automatización de pruebas de aplicaciones web.

Sahi sigue un enfoque de proxy externo sobre un navegador convencional.

En la actualidad da soporte, entre otros, a los navegadores Mozilla Firefox, Google Chrome y Microsoft Internet Explorer.

El funcionamiento de Sahi es el siguiente: cada vez que el navegador web solicita una nueva página, el proxy captura esta página y añade código JavaScript en ella para poder monitorizar las acciones de usuario. La página que se carga en el navegador convencional, es una versión modificada de la página que envía el servidor web.

Además de la versión de código abierto, esta herramienta también ofrece una versión comercial denominada Sahi Pro.

Para implementar las tareas de automatización web, Sahi requiere de la ejecución de scripts desarrollados en JavaScript, que serán evaluados a través del intérprete Mozilla Rhino.

La Figura 26, ilustra un ejemplo de un script de Sahi que permite automatizar el proceso de identificación en un sitio web.

```
// function declaration
function login($usr, $pwd){
  _click(_link("Login"));
  _setValue(_textbox("username"), $usr);
  _setValue(_password("password"), $pwd);
  _click(_submit("Login"));
}

// function call
login("test", "secret");
```

Figura 26. Ejemplo de script en Sahi.

Para ello, en primer lugar, el script ejecuta un evento *click* sobre un enlace con el texto "*Login*". Tras emitir este evento, el componente navega hasta la página de autenticación. En esta página, es posible localizar y rellenar el formulario que tiene un campo de texto llamado "*username*" y un campo de tipo contraseña llamado "*password*". La última sentencia del script ejecuta un evento *click* sobre el botón, lo que permite enviar el formulario.

2.3.6 Selenium

Selenium [SLNM] es un conjunto de herramientas de código abierto, desarrolladas para la automatización de pruebas de aplicaciones web.

Selenium IDE proporciona una extensión del navegador web Mozilla Firefox, que permite grabar una secuencia de acciones sobre una aplicación web, y después ejecutarlas de manera automática sobre alguno de los navegadores soportados:

- Navegadores convencionales: Mozilla Firefox, Google Chrome, Internet Explorer, o PhantomJS [PHAJS] (basado en WebKit), etc.
- Navegadores a medida: HtmlUnit [HTMUNIT].

La Figura 27, muestra una secuencia que navega al artículo de la Wikipedia sobre "*Word Wide Web*", utilizando el complemento de Selenium para Firefox (la secuencia de navegación ejecuta las mismas acciones que el ejemplo de la Figura 25).

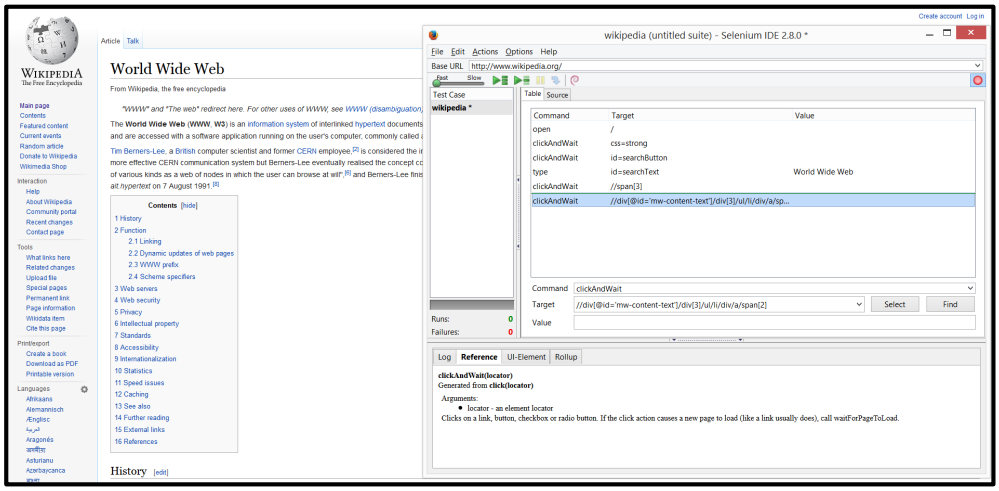


Figura 27. Ejemplo de secuencia de navegación web en Selenium.

En primer lugar, esta secuencia de navegación accede a la página de búsqueda del sitio web. A continuación, rellena el formulario con el término de búsqueda "*Word Wide Web*" y ejecuta un evento "*click*" sobre el botón de buscar. Por último, en la página de resultados, el último comando de la secuencia de navegación, selecciona el primero de ellos.

Otra de las herramientas que ofrece Selenium es *Selenium Remote Control*. Esta herramienta, permite controlar un navegador de manera local o remota utilizando para ello un servidor centralizado.

Este servidor es el encargado de ejecutar, controlar y cerrar los navegadores sobre los que se ejecutarán los comandos de las secuencias de navegación (por ejemplo, Internet Explorer, Mozilla Firefox o Safari). De manera adicional, este servidor actúa como proxy de las peticiones HTTP que recibe el navegador.

Selenium proporciona un API cliente, que permite acceder de forma programática al servidor. Este cliente, se puede implementar en una gran variedad de lenguajes de programación.

Selenium también ofrece un API de programación para incrustar un navegador web dentro de una aplicación desarrollada en alguno de los principales lenguajes (por ejemplo, Java, C#, Python, Ruby, etc.). Este API se denomina *Selenium WebDriver*.

```
using OpenQA.Selenium;
using OpenQA.Selenium.Firefox;

// Requires reference to WebDriver.Support.dll
using OpenQA.Selenium.Support.UI;

class GoogleSuggest
{
    static void Main(string[] args)
    {
        // Create a new instance of the Firefox driver.

        // Notice that the remainder of the code relies on the interface,
        // not the implementation.

        // Further note that other drivers (InternetExplorerDriver,
        // ChromeDriver, etc.) will require further configuration
        // before this example will work. See the wiki pages for the
        // individual drivers at http://code.google.com/p/selenium/wiki
        // for further information.
        IWebDriver driver = new FirefoxDriver();

        //Notice navigation is slightly different than the Java version
        //This is because 'get' is a keyword in C#
        driver.Navigate().GoToUrl("http://www.google.com/");

        // Find the text input element by its name
        IWebElement query = driver.FindElement(By.Name("q"));

        // Enter something to search for
        query.SendKeys("Cheese");

        // Now submit the form. WebDriver will find the form for us from the element
        query.Submit();

        // Google's search is rendered dynamically with JavaScript.
        // Wait for the page to load, timeout after 10 seconds
        WebDriverWait wait = new WebDriverWait(driver, TimeSpan.FromSeconds(10));
        wait.Until((d) => { return d.Title.ToLower().StartsWith("cheese"); });

        // Should see: "Cheese - Google Search"
        System.Console.WriteLine("Page title is: " + driver.Title);

        //Close the browser
        driver.Quit();
    }
}
```

Figura 28. Ejemplo de código de Selenium WebDriver en C#.

La Figura 28, detalla un ejemplo de código escrito en el lenguaje C# que permite realizar una búsqueda en Google utilizando *Selenium Web Driver* con el navegador Mozilla Firefox.

En este ejemplo, se crea en primer lugar la instancia del navegador y una vez hecho esto, se navega a la página de búsqueda. A continuación, se localiza el campo de texto (identificado con el atributo "*name*" con valor "*q*"), se establecen las palabras clave de búsqueda, y en último lugar, se realiza el envío del formulario.

2.3.7 SmartBookmarks

SmartBookmarks [HM07] es una herramienta de grabación de marcadores (*bookmarks*).

Smart Bookmarks considera que una secuencia de navegación (denominada “smart bookmark”) consiste en una URL de inicio y una secuencia de comandos que permiten alcanzar una página web particular, o un estado de una aplicación web.

Smart Bookmarks se ha desarrollado como una extensión del navegador web Mozilla Firefox.

2.3.8 Wargo

Wargo [PRAHV02] es un sistema que permite crear envoltorios sobre fuentes web de forma semi-automática.

Este sistema, incluye una herramienta para crear secuencias de navegación, de forma gráfica.

Las secuencias de navegación se basan en un lenguaje propietario denominado NSEQL (*Navigation SEQuence Language*).

Wargo incluye un módulo de navegación, capaz de interpretar programas NSEQL y ejecutarlos sobre un control de navegación basado en Microsoft Internet Explorer.

Este módulo de navegación, es el encargado de acceder a todas las páginas HTML y una vez que descarga los documentos HTML, obtiene su código fuente para enviárselo al módulo de extracción de datos.

2.3.9 PhantomJS

PhantomJS [PHAJS] es una herramienta de libre distribución basada en el motor de renderización WebKit [WEBKIT], que proporciona un navegador sin interfaz de usuario (aunque carece de interfaz gráfica, la renderización del documento se realiza de manera íntegra).

El motor de renderización WebKit, es el que utilizan navegadores tan populares como Apple Safari [APPSAF] o Google Chrome [GOOCHR] (en sus primeras versiones).

Entre los casos de uso de PhantomJS, se pueden destacar los siguientes:

- Pruebas de unidad de aplicaciones web.
- Captura de documentos HTML (en formato texto, imagen, etc.).
- Monitorización y análisis del tráfico de red, durante la carga de documentos HTML, etc.

Para acceder y manipular el componente de navegación basado en WebKit, esta herramienta ofrece un ejecutable de línea de comandos que permite la ejecución de scripts escritos en lenguaje JavaScript.

Este programa funciona según la siguiente sintaxis:

```
phantomjs [options] somescript.js [arg1 [arg2 [...]]]
```

Figura 29. Sintaxis de línea de comandos de PhantomJS.

```
var page = require('webpage').create();
console.log('The default user agent is ' + page.settings.userAgent);
page.settings.userAgent = 'SpecialAgent';
page.open('http://www.httpuseragent.org', function(status) {
  if (status !== 'success') {
    console.log('Unable to access network');
  } else {
    var ua = page.evaluate(function() {
      return document.getElementById('myagent').textContent;
    });
    console.log(ua);
  }
  phantom.exit();
});
```

Figura 30. Ejemplo de script ejecutado con PhantomJS.

Cuando se ejecuta sin ningún argumento, se abre un terminal interactivo.

Desde estos scripts se pueden cargar páginas web, manipular el árbol DOM de los documentos construidos, etc.

La Figura 30, detalla un ejemplo de un script de PhantomJS que accede a un sitio web (*httpuseragent.org*) y una vez que la página se ha cargado con éxito, el script obtiene una referencia a un elemento de la página, identificado con el valor "*myagent*".

2.3.10 WebMacros

WebMacros [SKC01] es un sistema basado en un intermediario o proxy para la automatización de secuencias de navegación web.

El modelo utilizado en WebMacros está basado en reescribir el código HTML de las páginas a las que se va accediendo.

2.3.11 WebVCR

Para finalizar, WebVCR [AFKL00] es un sistema que permite guardar y utilizar marcadores dinámicos, es decir, marcadores para acceder a páginas que no se pueden alcanzar, de forma directa, invocando una URL, sino que son necesarios varios pasos de navegación para llegar a ellas.

2.4 NAVEGADORES WEB A MEDIDA

Los navegadores desarrollados a medida, son componentes de navegación que utilizan como base la arquitectura de alto nivel de los navegadores tradicionales, pero están desarrollados de forma específica, para la ejecución de secuencias de navegación en los entornos de automatización web.

2.4.1 Arquitectura de referencia de los navegadores a medida

Los navegadores a medida, suelen simular el comportamiento de los navegadores tradicionales y están desarrollados con dos objetivos principales:

1. El primero de estos objetivos consiste en alcanzar una simulación del navegador convencional lo más precisa posible.

   Si este objetivo no se consigue, es posible que la ejecución de la secuencia de navegación dé lugar a una página web incorrecta o incompleta.

2. El segundo objetivo que se busca durante el desarrollo de un navegador a medida, consiste en alcanzar un alto nivel de eficiencia.

   La eficiencia es muy importante en los navegadores tradicionales, pero en los entornos de automatización web, es un aspecto crítico en la mayoría de las ocasiones.

   Por ejemplo, algunos escenarios de automatización web requieren de una respuesta en tiempo real y en otros casos, se requiere un alto nivel de concurrencia, con una gran cantidad de navegadores ejecutándose en paralelo. Es por ello que en estos entornos, el uso de CPU y de memoria debe de minimizarse y optimizarse lo máximo posible.

La Figura 31, ilustra la arquitectura típica de los navegadores a medida. Una de las principales diferencias con los navegadores convencionales, consiste en que los navegadores a medida no necesitan una interfaz de usuario en la que visualizar la página HTML que está cargada.

Los navegadores a medida, no están desarrollados para ser utilizados por personas, por lo que la visualización gráfica no es necesaria para la correcta ejecución de la secuencia de navegación.

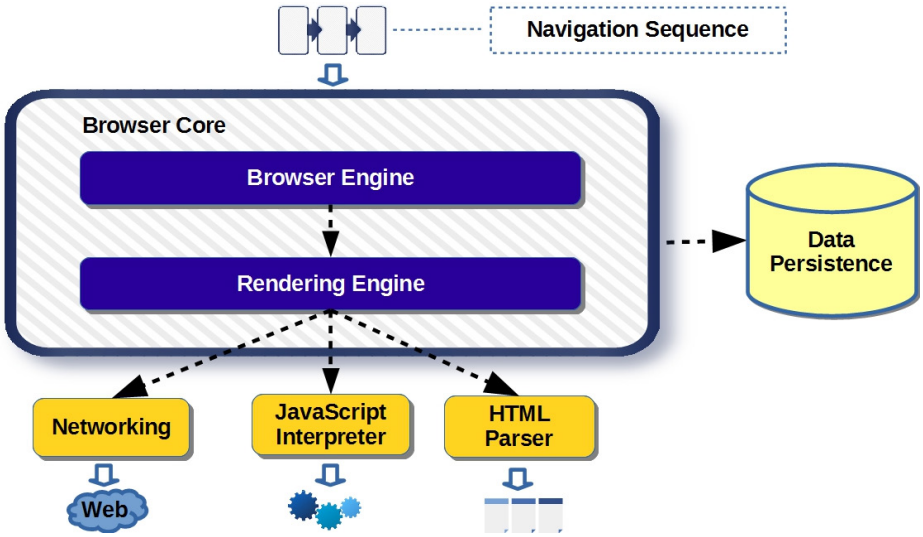


Figura 31. Arquitectura de los navegadores a medida.

Esto aumentará la eficiencia de estos componentes porque, en muchos casos, se evita la construcción del esquema de diseño de la página con la información de visualización y se evita también el renderizado de la página en la ventana del navegador.

Por lo tanto, el motor de renderización de un navegador desarrollado a medida se puede simplificar para evitar algunas tareas, que sólo son imprescindibles en los navegadores convencionales.

En la arquitectura de los navegadores a medida, el motor de navegación (*browser engine*) es también el punto de entrada que permite acceder al motor de renderización (*rendering engine*), pero en este caso, las acciones a ejecutar no provienen de la interfaz de usuario, sino que provienen de la secuencia de navegación web que se va a ejecutar.

Los distintos comandos que forman parte de una secuencia de navegación web, representan acciones que un usuario humano podría haber ejecutado con anterioridad sobre un navegador real, por ejemplo, navegar a una URL o emitir un evento click sobre un elemento del árbol DOM de la página.

2.4.2 Motor de renderización de los navegadores a medida

De forma similar a cómo sucede en los navegadores convencionales, el motor de renderización en los navegadores a medida también representa el núcleo central dentro del componente de navegación, al ser el encargado del procesamiento de los contenidos HTML descargados de la Web.

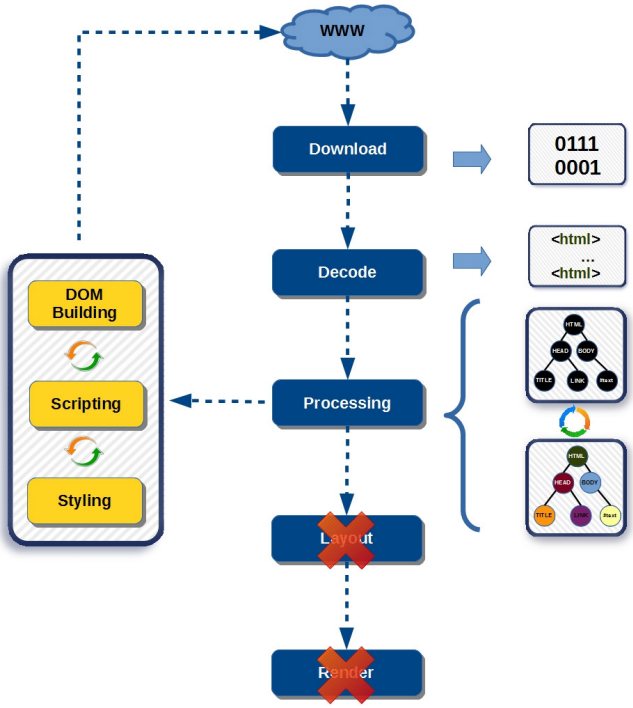


Figura 32. Etapas del motor de renderizado en un navegador a medida.

La Figura 32, detalla etapas por las que pasa el motor de renderización de un navegador a medida durante el procesamiento de un documento HTML.

En la figura, se muestran también las etapas que no son necesarias para la ejecución de la secuencia de navegación (construcción del esquema de diseño y renderización del documento en la pantalla) y que por lo tanto, al no ejecutarse, aumentará la eficiencia en la ejecución cuando se utilizan estos componentes.

2.5 AUTOMATIZACIÓN WEB BASADA EN NAVEGADORES DESARROLLADOS A MEDIDA

En el siguiente apartado, se detallan las principales herramientas de automatización web basadas en componentes de navegación desarrollados ad-hoc.

2.5.1 HTMLUnit

HtmlUnit [HTMUNIT] es una herramienta de código abierto, utilizada sobre todo para la automatización de pruebas de aplicaciones web.

Este sistema, proporciona un navegador sin interfaz gráfica con soporte JavaScript y con las siguientes funcionalidades principales:

1. Soporte para los protocolos HTTP y HTTPS y soporte de cookies.
2. Soporte de hojas de estilo CSS.
3. Capacidad para emular el comportamiento del navegador Mozilla Firefox y del navegador Microsoft Internet Explorer.
4. Soporte para proxy y para autenticación básica y NTLM.
5. Soporte para manejar el árbol DOM de las páginas web :
    - Permite realizar búsquedas de elementos por diferentes criterios.
    - Permite modificar los valores de los atributos de los nodos.
    - Permite realizar el envío de formularios.
    - Permite seguir enlaces.
    - Etc.

HtmlUnit ha sido desarrollado en Java y ofrece un API de alto nivel que permite su utilización dentro de cualquier aplicación escrita en este lenguaje.

En la Figura 33, se muestran los componentes principales que forman parte de la arquitectura de HtmlUnit:

1. Core: objetos del navegador que implementan el motor de renderización y el motor de navegación de HtmlUnit.
2. Mozilla Rhino [MOZRHN]: motor de ejecución de JavaScript.
3. CSS Parser [CSSPAR]: librería utilizada para procesar las hojas de estilo CSS.
4. Apache HttpClient [APHTTPC4]: librería utilizada para la ejecución de peticiones HTTP.

El funcionamiento del motor de renderización de HtmlUnit, está basado en la utilización de dos hilos para cada uno de los navegadores.

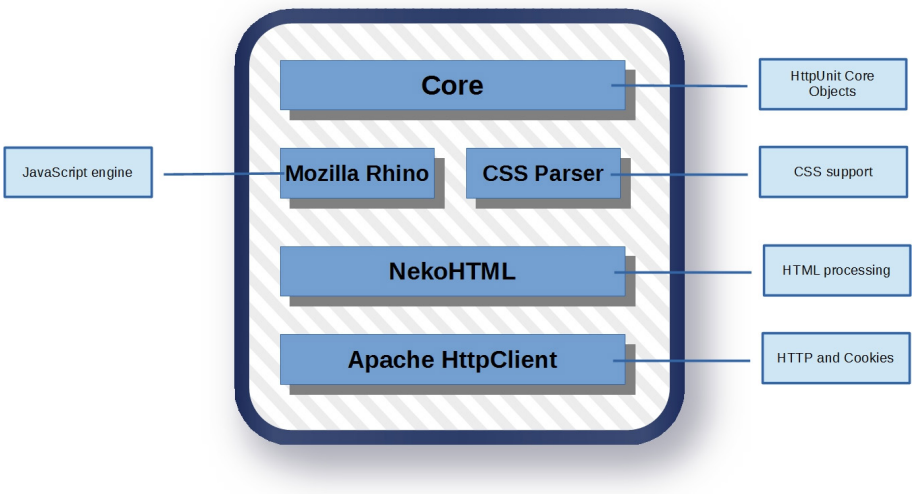


Figura 33. Módulos que componen la arquitectura de HtmlUnit.

```
public void submittingForm() throws Exception {
    final WebClient webClient = new WebClient();

    // Get the first page
    final HtmlPage page1 = webClient.getPage("http://some_url");

    // Get the form that we are dealing with and within that form,
    // find the submit button and the field that we want to change.
    final HtmlForm form = page1.getFormByName("myform");

    final HtmlSubmitInput button = form.getInputByName("submitbutton");
    final HtmlTextInput textField = form.getInputByName("userid");

    // Change the value of the text field
    textField.setValueAttribute("root");

    // Now submit the form by clicking the button and get back the second page.
    final HtmlPage page2 = button.click();

    webClient.closeAllWindows();
}
```

Figura 34. Ejemplo de envío de formulario en HtmlUnit.

En uno de estos hilos, reside el proceso principal. En este proceso se ejecutarán, de manera secuencial, la mayoría de las acciones involucradas en la construcción de los documentos HTML.

El segundo hilo, se crea de forma automática durante la inicialización de cada navegador y su tarea consiste en ejecutar el código JavaScript contenido en la página HTML (también de manera secuencial).

Con este modelo de funcionamiento, utilizando dos hilos, es posible detener la evaluación de JavaScript pasado un tiempo máximo de ejecución.

La Figura 34, detalla un ejemplo de envío de un formulario utilizando el API que proporciona HtmlUnit.

Este código de ejemplo, accede a una página web y una vez que la ha cargado, obtiene un formulario identificado con el nombre "*myform*". A continuación, establece un valor sobre un campo de texto con nombre "*userid*" y en último lugar,

ejecuta una acción "*click*" sobre un botón del formulario, identificado con el nombre "*submitbutton*".

2.5.2 EnvJS

EnvJS [ENVJS] es una herramienta que proporciona un componente de navegación sin interfaz de usuario, accesible a través de un entorno de ejecución de JavaScript.

Esta herramienta, está desarrollada en Java y utiliza por defecto el motor de ejecución de JavaScript Mozilla Rhino.

EnvJS permite la ejecución de scripts de diversas maneras:

1. A través de línea de comandos.
2. A través de una consola interactiva.
3. Incrustando los scripts dentro de una aplicación Java.

La Figura 35, detalla un ejemplo de script ejecutable utilizando EnvJS, junto con la popular librería JavaScript JQuery [JQUERY].

En este ejemplo, se realiza una navegación a la dirección "*http://localhost:8080*" (línea 41) y una vez que ha finalizado de cargarse la página, se ejecuta la función definida a partir de la línea 24.

Esta función extrae todos los enlaces, junto con información adicional (para ello, utiliza la función "*scrape*" definida a partir de la línea 4).

Para cada uno de los enlaces identificados, se repite el mismo proceso de manera recursiva.

```
load('dist/env.rhino.js');
load('plugins/jquery.js');

function scrape(url, links){
    // Scrape text from current document
    var data = {
        $id: encodeURIComponent(url),
        url: url,
        full_text: $(document.body).text(),
        title: document.title,
        headings: $('h1, h2, h3, h4, h5, h6').text(),
        description: $('meta[name=description]').attr('content'),
        keywords: $('meta[name=keywords]').attr('content').split(',')
    };
    // Find all the links, but don't include any we already have in our link array
    $('a[href]').each(function(){
        var href = $(this).attr('href');
        if($.inArray(href, links) === -1){
            links.push(href);
        }
    });
}

$(function(){
    // Create an array which we'll use to store links to crawl
    var links = [];
    // Index this document
    scrape(document.location.toString(), links);
    // Now crawl links
    for(var i = 0; i < links.length; i++){
        try{
            // Replaces this document with the document from the link
            document.location = Envjs.uri(links[i]);
            scrape(links[i], links);
        }catch(e){
            console.log('failed to load %s \n %s', links[i], e);
        }
    }
});

window.location = 'http://localhost:8080/';
```

Figura 35. Ejemplo de script de EnvJS.

2.5.3 ZombieJS

ZombieJS [ZOMJS] es un componente desarrollado para la ejecución de pruebas de unidad de aplicaciones web.

Esta herramienta está basada en Node.js [NODEJS], plataforma que permite la ejecución de JavaScript en el lado servidor (y construida a su vez a partir del motor de JavaScript V8 [V8], utilizado en diversos navegadores convencionales, como por ejemplo Google Chrome).

```
browser.visit('http://localhost:8080/form', function() {
    browser
            .fill('Name', 'Jose Losada')
            .select('Born', '1978')
            .check('Agree with terms and conditions')
            .pressButton('Submit', function() {
                assert.equal(browser.location.pathname, '/success');
                assert.equal(browser.text('#message'),
                        'Thank you for submitting this form!');
            });
});
```

Figura 36. Ejemplo de script en ZombieJS.

La Figura 36, muestra un ejemplo de un script que se ejecuta con ZombieJS y que permite acceder a un formulario, para establecer dos campos de búsqueda con el nombre y la fecha de nacimiento.

A continuación, el script selecciona una casilla de verificación y realiza el envío del formulario.

En último lugar, se realiza una comprobación para validar que el formulario se ha enviado de forma correcta.

2.5.4 Jaunt

Jaunt [JAUNT] es una herramienta de automatización web desarrollada en Java y que proporciona un componente de navegación muy sencillo, sin soporte de JavaScript.

Entre las principales funcionalidades de Jaunt, destacan las siguientes:

1. Soporte para HTML y XML.
2. Soporte de HTTP y HTTPS, proxy y autenticación básica.
3. Manipulación del árbol DOM.
4. Descarga y envío de ficheros, etc.

Para dar soporte a comandos de manipulación de formularios, esta herramienta permite identificar los campos de entrada utilizando los elementos visuales (etiquetas) que se encuentran más próximos.

Para acceder a otros elementos del árbol DOM, Jaunt ofrece un API propietario y no soporta ni XPath ni selectores CSS.

```
try{
  UserAgent userAgent = new UserAgent();
  userAgent.visit("http://jaunt-api.com/examples/login.htm");

  userAgent.doc.fillout("Username:", "tom");
  userAgent.doc.fillout("Password:", "secret");
  userAgent.doc.choose(Label.RIGHT, "Remember me");
  userAgent.doc.submit();
  System.out.println(userAgent.getLocation());
}
catch(JauntException e){
  System.err.println(e);
}
```

Figura 37. Ejemplo de secuencia de navegación en Jaunt.

La Figura 37, detalla un ejemplo de una secuencia de navegación que utiliza el componente de navegación de Jaunt (llamado *UserAgent*) para acceder a una página web, rellenar y enviar un formulario.

2.5.5 Twill

Twill [TWILL] es una herramienta de línea de comandos desarrollada en Python que permite ejecutar tareas de automatización web.

Esta herramienta ofrece una consola (*twill-sh*), sobre la que se pueden ejecutar las distintas acciones de navegación.

Además, también proporciona un mecanismo para ejecutar pruebas de carga, permitiendo ejecutar un mismo script un número predefinido de veces.

Entre los comandos que se pueden incluir en un script de Twill, destacan los comandos de navegación (por ejemplo *go*, para navegar a una URL), comandos de aserción para detectar elementos presentes en la página web (por ejemplo, a través de expresiones regulares), comandos para manipular formularios (por ejemplo, *formvalue* permite establecer el valor de un campo), etc.

Twill ofrece un mecanismo para desarrollar módulos específicos para las diferentes tareas de automatización web.

La Figura 38, muestra un ejemplo de un script desarrollado en Twill.

Este ejemplo utiliza comandos desarrollados, de forma específica, para una tarea de automatización web que accede a una lista de correo del sitio web *sourceforge.com* y elimina los mensajes contenidos en una carpeta determinada (este módulo de comandos específicos se denomina *mailman_sf*).

```
# load in the mailman_sf extensions module
extend_with mailman_sf

# Navigate to mailing list
go https://lists.sourceforge.net/lists/admindb/pywx-announce

# fill out the page with the list password.
getpassword "Enter list password: "
formvalue 1 adminpw __password__
submit 0
code 200

# if there aren't any messages on the page, exit.
exit_if_empty

# if not empty, discard all messages & submit
discard_all_messages
submit 0
```

Figura 38. Ejemplo de secuencia de navegación en Twill.

```
var browser = new Browser();
browser.UserAgent = "Mozilla/5.0 (Windows; U; Windows NT 6.1; en-US) ...";
browser.Navigate("http://github.com/");
var loginLink = browser.Find("a", FindBy.Text, "Login");
if(loginLink.Exists) {
    loginLink.Click();
    browser.Find("login_field").Value = "youremail@domain.com";
    browser.Find("password").Value = "yourpassword";
    browser.Find(ElementType.Button, "name", "commit").Click();
    if(browser.ContainsText("Your Repositories")) {
        for each(var item in browser.Select("div.news .title")) {
            browser.Log("* " + item.Value);
        }
    }
}
```

Figura 39. Ejemplo de secuencia de comandos en SimpleBrowser.

2.5.6 SimpleBrowser

SimpleBrowser [SIMBRO] es una herramienta de automatización web desarrollada en .NET, que proporciona un componente de navegación muy ligero sin soporte JavaScript.

Entre sus funcionalidades principales, destaca la capacidad para manipular formularios y las cabeceras de las peticiones HTTP.

En la Figura 39, se puede observar un ejemplo de una secuencia de comandos de SimpleBrowser que navega a una página web, ejecuta el proceso de identificación en la cuenta del usuario y muestra una serie de elementos contenidos en el documento HTML.

2.6 DISCUSIÓN Y CONCLUSIONES

Una vez que se han expuesto las principales características de los navegadores convencionales y de los navegadores a medida, es posible realizar una discusión sobre las ventajas e inconvenientes de cada una de estas dos aproximaciones, a la hora de construir un sistema de automatización web.

2.6.1 Ventajas e inconvenientes de los sistemas de automatización web basados en navegadores convencionales

La ventaja principal de los sistemas de automatización web basados en navegadores convencionales, reside en que su desarrollo es relativamente sencillo.

La implementación de los distintos comandos que forman parte de una secuencia de navegación web, se traduce, de forma casi directa, en llamadas del API de alto nivel que proporcionan estos navegadores, y por lo tanto, es el propio motor de renderización del navegador convencional, el que se encarga de ejecutar todas las acciones asociadas a cada llamada del API.

Como se detalla en el apartado 2.2.2, el proceso de renderización en los navegadores convencionales es un proceso costoso, que involucra no sólo la construcción del árbol DOM, sino también la construcción de otras estructuras adicionales, con el objetivo final de mostrar los documentos HTML renderizados en la ventana.

Los navegadores tradicionales, son aplicaciones desarrolladas para ser utilizadas por usuarios humanos y todos los navegadores analizados siguen la arquitectura de referencia detallada en el apartado 2.2 (con pequeñas variaciones).

Algunos de los elementos que forman parte de esta arquitectura y algunas de las etapas por las que pasa su motor de renderización durante el procesado de los documentos HTML (detalladas en el apartado 2.2.3), no son necesarios cuando se ejecutan tareas de automatización web.

Es por ello, que el principal inconveniente de estos sistemas reside en su bajo nivel de eficiencia, y por lo tanto, puede no ser la solución idónea para ejecutar tareas de automatización web, que requieren respuestas en tiempo real o tareas de automatización web, que requieren de un gran número de navegadores ejecutándose de forma simultánea.

2.6.2 Ventajas e inconvenientes de los sistemas de automatización web basados en navegadores a medida.

Los navegadores a medida, no están diseñados para ser utilizados por usuarios finales y por lo tanto no necesitan interfaz gráfica.

Esto se traduce en una arquitectura más sencilla que la de los navegadores convencionales.

Esta arquitectura, detallada en el apartado 2.4.1, no incluye algunos subsistemas de los navegadores convencionales: la interfaz gráfica y la capa de visualización.

Como consecuencia, el motor de renderización de los navegadores a medida (detallado en el apartado 2.4.2) puede evitar algunos pasos a la hora de procesar los documentos HTML: la fase de construcción del árbol de visualización y la fase de renderización.

Por lo tanto, la ventaja principal de los sistemas de automatización web basados en navegadores a medida reside en que, en la mayoría de los escenarios, mejoran la eficiencia de los navegadores convencionales, al evitar o simplificar algunas de las acciones que se ejecutan en éstos.

Por otro lado, el proceso de desarrollo de un navegador a medida es un proceso complejo, que requiere de un gran esfuerzo para dar soporte a todas las tecnologías y funcionalidades disponibles hoy en día, en los navegadores convencionales.

Es por ello que algunos sistemas de automatización web, proporcionan componentes de navegación muy simplificados (por ejemplo, sin soporte para el lenguaje JavaScript) que sólo son útiles en tareas de automatización que acceden a sitios web sencillos. Este es el caso de Jaunt y SimpleBrowser, comentados en los apartados 2.5.4 y 2.5.6.

Otros sistemas de automatización web basados en navegadores a medida, soportan funcionalidades avanzadas de los navegadores tradicionales (sobre todo, soporte para el lenguaje JavaScript y soporte para hojas de estilo CSS). Entre estos sistemas destaca HtmlUnit, explicado en el apartado 2.5.1.

No obstante, el comportamiento de estos sistemas a la hora de procesar los documentos HTML, es muy similar al de los navegadores convencionales:

- El flujo de procesamiento es secuencial (aunque algunas tareas se pueden ejecutar en hilos auxiliares).
- El documento HTML se procesa de manera íntegra.
- Aunque el componente de navegación carece de interfaz de usuario, el procesamiento de las hojas de estilo CSS se realiza en su totalidad.

2.6.3 Mejoras en los sistemas de automatización web basados en navegadores a medida

A continuación, se explican algunos puntos en los cuales es posible desarrollar mejoras y optimizaciones sobre los sistemas actuales de automatización web, basados en componentes de navegación desarrollados a medida.

Estas mejoras se pueden englobar en dos grandes grupos, que se detallan a continuación.

1. En primer lugar, mejoras basadas en el hecho de que las secuencias de navegación se conocen a priori y siguen una estructura repetitiva.
2. En segundo lugar, mejoras basadas en que la interfaz de usuario no es necesaria en los componentes de navegación a medida, desarrollados de forma específica, para tareas de automatización web.

A continuación, se detallan las posibles optimizaciones que se pueden desarrollar teniendo en cuenta estas peculiaridades de los entornos de automatización web.

2.6.3.1 Mejoras basadas en el hecho de que las secuencias de navegación web son conocidas a priori

En los entornos de automatización web, una misma secuencia de navegación suele ejecutarse en repetidas ocasiones, variando en cada una de ellas los parámetros de entrada según los criterios de búsqueda establecidos en cada momento.

Aunque los parámetros de entrada pueden ser diferentes en cada ejecución de la misma secuencia, las páginas que atraviesa suelen tener la misma estructura y además, es frecuente que estas páginas contengan muchos elementos prescindibles e irrelevantes para la ejecución de la secuencia.

Esto significa que muchos elementos de las páginas HTML, no son necesarios para que la secuencia funcione de manera correcta y un ejemplo que ilustra este escenario son los anuncios de publicidad.

Si se eliminasen estos anuncios de la página HTML, en la práctica totalidad de las situaciones, la secuencia de navegación seguiría funcionando, pero el proceso de carga de la página HTML sería mucho más rápido, produciéndose mejoras sustanciales en el uso de recursos:

1. Menos uso de ancho de banda: el anuncio publicitario no se descargaría de Internet.
2. Menos uso de CPU y de memoria: el tiempo de procesamiento del anuncio se evitaría, así como el uso de memoria asociado.

Además de los anuncios publicitarios, en las páginas HTML suelen existir otros elementos que en muchos casos tampoco son necesarios para la correcta ejecución de la secuencia de navegación. Este es el caso de encabezados y pies de página, menús, barras laterales, etc.

En la Figura 40, se muestra un ejemplo de este escenario sobre un sitio web real, en este caso, la popular tienda de comercio electrónico, Amazon.

Si un componente de navegación web ejecutase la siguiente secuencia de acciones:

1. Navegar al sitio web de la tienda de comercio electrónico.
2. Localizar el formulario de búsqueda (se encuentra en la parte superior de la pantalla, identificado con la etiqueta "*Search*").
3. Rellenar el campo de texto con las palabras clave.
4. Emitir un evento *click* sobre el botón de buscar (localizado justo después del campo de texto, e identificado con la etiqueta "*Go*").

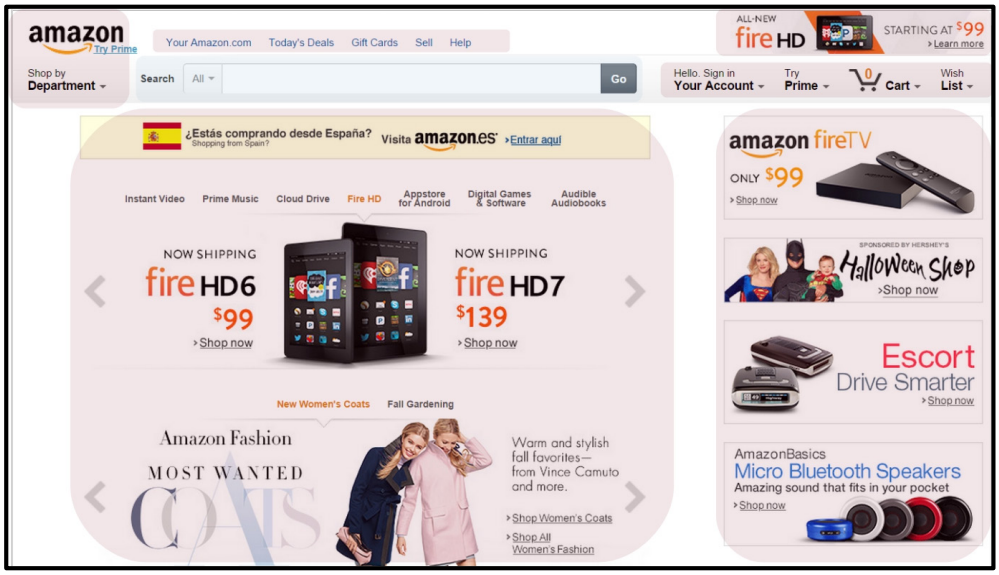


Figura 40. Componentes de la página web de amazon.com.

Los elementos resaltados en la figura no se verían involucrados, o lo que es lo mismo, no serían necesarios para la correcta ejecución de la secuencia de navegación web.

Como la misma secuencia se ejecuta de forma repetitiva, es posible realizar un análisis durante una primera ejecución para detectar los elementos que no son necesarios.

Si un navegador a medida es capaz de identificar los elementos que no se utilizan durante la ejecución de la secuencia, también sería capaz de construir una página más pequeña sólo con los elementos necesarios, descartando de forma automática todos aquellos fragmentos que han sido detectados como irrelevantes.

La mejora potencial en estos casos sería importante, sobre todo en aquellas páginas en las que los elementos involucrados en la ejecución de la secuencia, representan un porcentaje muy pequeño con respecto del total de nodos contenidos en el árbol DOM.

Este proceso de detección de elementos necesarios y no necesarios, debe tener en cuenta también la evaluación de JavaScript, dado que muchos de los nodos presentes en la página son accedidos por el código JavaScript, para realizar diferentes operaciones sobre ellos.

Por último, si el navegador a medida es capaz de analizar la ejecución del código JavaScript para detectar los elementos que se usan en cada uno de los scripts definidos en la página (variables, nodos del árbol DOM, etc.), y es capaz de detectar las distintas interacciones que se producen entre todos ellos, también sería capaz de determinar el orden de ejecución de estos scripts, permitiendo ejecutar en paralelo scripts que no tengan dependencias entre ellos (por ejemplo, scripts que no comparten variables, etc.)

2.6.3.2 Mejoras basadas en que la visualización de la página no es necesaria en los navegadores a medida

Los navegadores a medida, no están diseñados para ser utilizados por personas, con lo que la visualización de la página HTML renderizada, no es necesaria para la correcta ejecución de la secuencia de navegación web.

Esta característica, ya está disponible en muchos de los sistemas actuales, los cuales proporcionan un componente de navegación sin interfaz gráfica y sin capacidades de renderización.

No obstante, la información de visualización puede ser accedida durante la ejecución del código JavaScript y por lo tanto, muchos de estos sistemas de automatización, aunque no proporcionan interfaz de usuario, sí dan soporte a las hojas de estilo CSS, realizando todos los cálculos necesarios para obtener los atributos de presentación, de cada uno de los nodos del árbol DOM.

Este funcionamiento actual, podría mejorarse de manera considerable, para evitar muchos de estos cálculos.

Esta mejora, consistiría en que el componente de navegación podría calcular la información de visualización (contenida en las hojas de estilo CSS), sólo cuando estos atributos fuesen requeridos durante la evaluación del código JavaScript.

Con ello, estos cálculos se realizarían, de forma exclusiva, para los elementos del árbol DOM para los que fuese imprescindible hacerlo, evitándose en aquellos nodos en los que sus propiedades de visualización no se utilicen durante la evaluación de JavaScript.

Se trataría, en este caso, de una evaluación bajo demanda y se ejecutaría sólo para un número muy reducido de nodos del árbol DOM.

3 DESCRIPCIÓN DE LAS TÉCNICAS DE OPTIMIZACIÓN PARA LA EJECUCIÓN EFICIENTE DE SECUENCIAS DE NAVEGACIÓN WEB

En este capítulo, se describen en detalle las distintas técnicas de optimización propuestas en este trabajo.

En primer lugar, se detallan las técnicas basadas en un análisis pormenorizado de una ejecución inicial de la secuencia de navegación.

Durante esta primera ejecución, se obtendrá información que será utilizada de forma posterior, para optimizar la ejecución de las siguientes ejecuciones de la misma secuencia.

Estas técnicas de optimización son las siguientes: por un lado, la construcción de un árbol DOM minimizado que contendrá solo los fragmentos de la página web que son estrictamente necesarios para el correcto funcionamiento de la secuencia de navegación web y por otro lado, la ejecución en paralelo de aquellos scripts contenidos en el documento HTML, que no tienen dependencias entre ellos.

En segundo lugar, se describe la técnica de optimización que explota la ausencia de interfaz gráfica en los navegadores desarrollados a medida.

Esta técnica lleva por nombre evaluación de estilos CSS bajo demanda y consiste en la ejecución parcial del cálculo de la información de visualización.

Este cálculo, se realizará sólo cuando los atributos de visualización sean imprescindibles para el correcto funcionamiento del código JavaScript contenido en el documento HTML.

3.1 CONSTRUCCIÓN DE UN ÁRBOL DOM MINIMIZADO DE LA PÁGINA HTML

Esta primera técnica de optimización, consiste en la construcción de una página HTML reducida, eliminando de ella, aquellos elementos que no son necesarios para que la secuencia de navegación web se ejecute de forma correcta.

Para ello, se definen dos fases de trabajo diferentes: fase de optimización y fase de ejecución.

Durante la fase de optimización, la secuencia se ejecuta una sola vez. Durante esta ejecución, el componente de navegación calcula qué nodos son necesarios para que la secuencia se ejecute con éxito y qué nodos se pueden descartar.

Esta información se almacena de manera persistente, para que pueda ser utilizada en las siguientes ejecuciones de la misma secuencia.

Por otro lado, la forma de representar la información que permite identificar los fragmentos de la página que no son necesarios, debe ser lo suficiente robusta como para que pequeños cambios en la página web, no afecten a la identificación de dichos fragmentos en las siguientes ejecuciones de la misma secuencia.

Es necesario hacer hincapié en que, una misma página puede sufrir pequeñas variaciones al ser cargada en momentos diferentes (por ejemplo, cuando cambia un anuncio de publicidad).

Del mismo modo, el componente de navegación también será capaz de calcular cuáles de los eventos que se disparan de forma automática al cargar cada una de las páginas, son necesarios para el correcto funcionamiento de la secuencia.

La Figura 41, muestra el árbol DOM de una página sencilla, que ilustra esta técnica con un ejemplo. En esta figura, cada uno de los nodos está representado con un círculo y las líneas continuas representan las relaciones padre-hijo.

Los manejadores de eventos están representados con rectángulos de líneas discontinuas (*onclick* y *onload*), justo al lado de los nodos sobre los que se establecen esos manejadores.

Las flechas con líneas discontinuas, se usan para indicar que un script define una función o variable JavaScript (marcado con <<*def*>>) y también para indicar que un manejador de eventos invoca una función definida en un nodo de tipo script (marcado con <<*calls*>>).

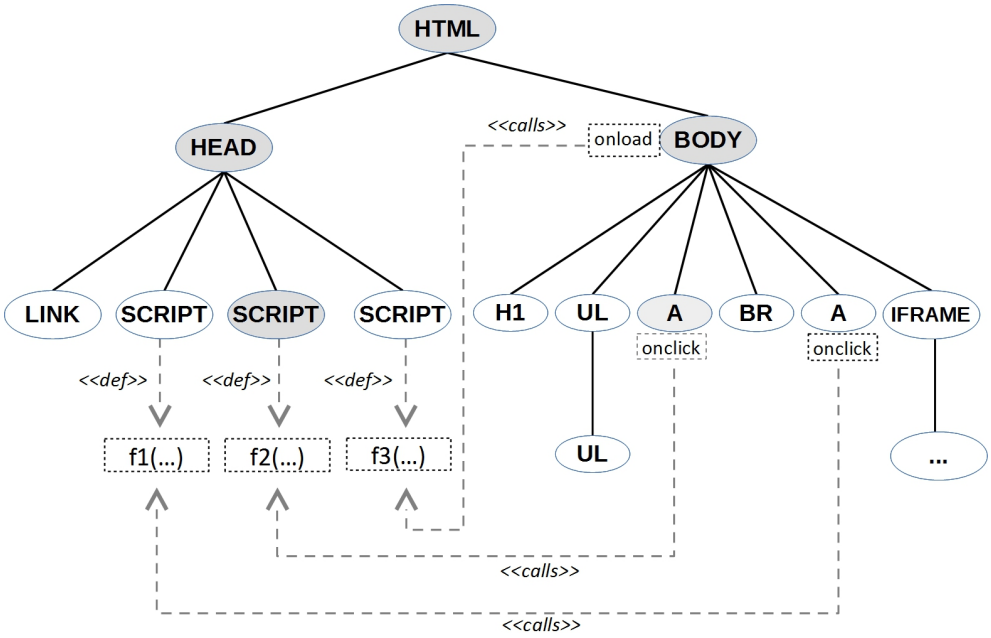


Figura 41. Ejemplo del árbol DOM de una página sencilla.

Sobre este árbol DOM, podría ejecutarse una secuencia de navegación con una única acción, que podría consistir en un clic sobre el primer nodo de tipo *A*.

Cuando se emite el evento *click*, se ejecuta el manejador de evento *onclick*. Este manejador invoca a la función *f2* y esta función navega a otra página web diferente (por ejemplo, mediante el siguiente código JavaScript: *window.location = 'http://acme.com';)*.

Los nodos marcados en gris, son los necesarios para simular el evento *click* y ejecutar de manera correcta la navegación a la página siguiente (a estos nodos los denominaremos "nodos relevantes").

En este ejemplo, el conjunto completo de nodos relevantes es el siguiente:

- El nodo *A* sobre el que se emite el evento *click* desde un comando de la secuencia de navegación.
- El nodo *SCRIPT* que define la función *f2* ejecutada por el manejador de evento *onclick*.
- Los ancestros del nodo *A* y del nodo *SCRIPT* (las reglas que han sido utilizadas para calcular los nodos relevantes se describen más adelante).

El resto de los nodos, pueden ser descartados (no cargados), sin que ello afecte a la ejecución de la secuencia (estos nodos reciben el nombre de "nodos irrelevantes").

Además, no es necesario disparar el evento automático *load* cuando se carga la página. Esto se debe a que, para reproducir la secuencia de forma correcta, no hace falta ejecutar el código JavaScript asociado al manejador del evento *onload*.

Esta técnica de optimización, producirá mejoras significativas en el rendimiento y en el uso de recursos:

1. Se producirá un ahorro de memoria porque se crearán muchos menos nodos, durante la construcción del árbol DOM.
2. Se utilizará menos tiempo de CPU porque no se ejecutarán los scripts innecesarios.

   En el ejemplo anterior, no es necesario ejecutar los nodos script que no se muestran en gris.
3. Se producirá un ahorro en ancho de banda porque no se realizarán las navegaciones que son innecesarias.

   En el ejemplo anterior, no se realizarán las navegaciones especificadas por los nodos *LINK* e *IFRAME*.

El problema principal para desarrollar esta técnica de optimización, reside en cómo calcular lo que en este trabajo se denomina "*dependencias entre nodos*".

En el ejemplo anterior, el segundo nodo *SCRIPT* (el que define la función *f2*) es una dependencia del nodo *A* cuando se emite el evento *click* sobre este nodo *A*.

Esto significa, que este nodo de tipo SCRIPT no se puede eliminar del árbol DOM dado que si fuese eliminado, desaparecería la función *f2* y la ejecución del manejador *onclick* del nodo *A* no se ejecutaría de forma correcta.

Es importante indicar que en el modelo DOM, los scripts son "cajas negras", y por tanto, dichas dependencias no pueden ser inferidas a priori.

Sin embargo, al utilizar un navegador desarrollado de manera específica para ejecutar estas tareas, es posible controlar el motor de ejecución de scripts a más bajo nivel y por lo tanto, es posible detectar las dependencias entre los nodos (que siempre permanecen ocultas, cuando se utiliza el API de alto nivel de los navegadores convencionales).

También es necesario resaltar el hecho de que las dependencias, pueden llegar a ser mucho más complejas que las descritas en el escenario sencillo del ejemplo anterior.

Por ejemplo, la emisión de un evento *click* sobre un enlace puede producir la ejecución de una función definida en un nodo script, y para que esta función se ejecute de forma correcta, puede ser necesaria la inicialización de una variable, y esta acción, puede realizarse en otro script diferente del árbol DOM.

Otro ejemplo, con mayor dificultad, sería aquel en el que el manejador de evento *load* del nodo *BODY* genera contenido dinámico, incluyendo un nodo *A* sobre el cual se podría emitir más adelante un evento *click*.

Podría ocurrir incluso que un script haga uso de elementos definidos en otro script, y este último, podría estar definido en un marco diferente (*IFRAME*).

En los siguientes apartados, se describe la manera de abordar todos estos escenarios.

3.1.1 Dependencias entre nodos del árbol DOM

Definición 1: se dice que existe una dependencia entre dos nodos *n1* y *n2* cuando el nodo *n2* es necesario para la ejecución correcta del nodo *n1*.

Se dice que el nodo *n2* es una dependencia del nodo *n1* y se denota *n1* → *n2*.

Las siguientes reglas definen este tipo de dependencias:

a) Si el código script de un nodo *s1* utiliza un elemento declarado en un nodo *SCRIPT s2* (por ejemplo, una función o una variable), entonces *s1* → *s2*.

   Justificación: para poder ejecutar el código de script del nodo *s1*, debe ejecutarse con anterioridad el nodo *s2*.

b) Si el código script de un nodo *s* utiliza un nodo *n*, entonces *s* → *n*.

   Justificación: para poder ejecutar el código script del nodo *s*, debe cargarse con anterioridad el nodo *n*.

   Por ejemplo, si *s* obtiene una referencia al nodo *A*, utilizando la función JavaScript *getElementById* del objeto *document*, y navega a la URL especificada en su atributo *href*, entonces no se podrá ejecutar el script *s* de forma correcta, si no se carga con anterioridad el nodo *A*.

c) Si el código script de un nodo *s* realiza una modificación en un nodo *n*, entonces *n* → *s*.

   Justificación: la acción realizada por *s* puede ser necesaria en una acción posterior sobre el nodo *n*.

   Por ejemplo, si *s* modifica el atributo *action* de un nodo *FORM* para establecer la URL destino, entonces no podrá realizarse el envío del formulario a no ser que *s* se ejecute antes.

Definición 2: se dice que existe una dependencia condicionada a que el evento *e* se dispare sobre el nodo *n*, entre dos nodos *n1* y *n2*, cuando el nodo *n2* es necesario para la ejecución correcta del nodo *n1*, cuando se dispara el evento *e* sobre el nodo *n*. Se denota como $n1 \rightarrow^{e/n} n2$.

Este tipo de dependencias, se definen mediante un conjunto de reglas análogas a las explicadas en la definición anterior, con la diferencia de que en este caso, involucran nodos que contienen manejadores de eventos:

1. Si el código script de un manejador de eventos *l* para el evento *e* sobre el nodo *n*, utiliza un elemento declarado en un nodo script *s* (por ejemplo, una función o una variable), entonces $n \rightarrow^{e/n} s$.

Justificación: si el evento *e* se emite sobre el nodo *n*, entonces se ejecutará el manejador de eventos *l*, el cual requiere que se ejecute antes el nodo SCRIPT *s*.

2. Si el código script de un manejador de eventos *l* para el evento *e* sobre el nodo *n1*, utiliza un nodo *n2*, entonces $n1 \rightarrow^{e|n1} n2$.

   Justificación: si se emite el evento *e* sobre *n1*, entonces se ejecutará el manejador del evento *l* y por tanto, se deberá cargar antes el nodo *n2*.

3. Si el código script de un manejador de eventos *l* para el evento *e* sobre el nodo *n1*, realiza una modificación en un nodo *n2*, entonces $n2 \rightarrow^{e|n1} n1$.

   Justificación: la acción realizada por *l*, puede ser necesaria para que *n2* sea utilizado de forma posterior.

   Por ejemplo, si *l* modifica el atributo *action* de un nodo *FORM*, para establecer la URL destino, no será posible enviar el formulario a no ser que antes se ejecute *l*. Debido a que *l* sólo se ejecutará si se dispara el evento *e* sobre *n1*, entonces *n1* es necesario.

Las siguientes propiedades de transitividad, son aplicables a las dependencias entre nodos (cada una de ellas, será explicada con un ejemplo en la Figura 42).

Propiedad 1: Si $n1 \rightarrow n2$ y $n2 \rightarrow n3$ entonces $n1 \rightarrow n3$.

El ejemplo de la Figura 42.a, muestra un fragmento del árbol DOM de una página en la que el código script del nodo *SCRIPT1* invoca a la función *f1*, definida en el nodo *SCRIPT2* (*SCRIPT1*→ *SCRIPT2*), y el código de la función *f1* invoca a la función *f2*, definida en el nodo *SCRIPT3* (*SCRIPT2*→ *SCRIPT3*).

Para una correcta ejecución del código script del nodo *SCRIPT1*, son necesarios tanto el segundo como el tercer nodo *SCRIPT*, por tanto ambos son dependencias de éste (*SCRIPT1*→ *SCRIPT3*).

Propiedad 2: Si $n1 \rightarrow^{e|n} n2$ y $n2 \rightarrow n3$ entonces $n1 \rightarrow^{e|n} n3$.

El ejemplo de la Figura 42.b, muestra un fragmento del árbol DOM de una página donde el manejador de evento *click* de un nodo *A* invoca a la función *f1*, definida en el nodo *SCRIPT* ($A \rightarrow^{click|A} SCRIPT$), y el código de la función *f1* utiliza el atributo *src* del nodo *IMG* (*SCRIPT* → *IMG*).

Para realizar un procesado correcto del nodo *A*, cuando se emite el evento *click* sobre él, son necesarios tanto el nodo *SCRIPT* como el nodo *IMG*, y por lo tanto, ambos son dependencias del nodo *A* ($A \rightarrow^{click|A} IMG$).

Propiedad 3: Si $n1 \rightarrow n2$, y $n3 \rightarrow n2$, porque *n2* es un nodo script que realiza una modificación en *n3*, entonces $n3 \rightarrow n1$.

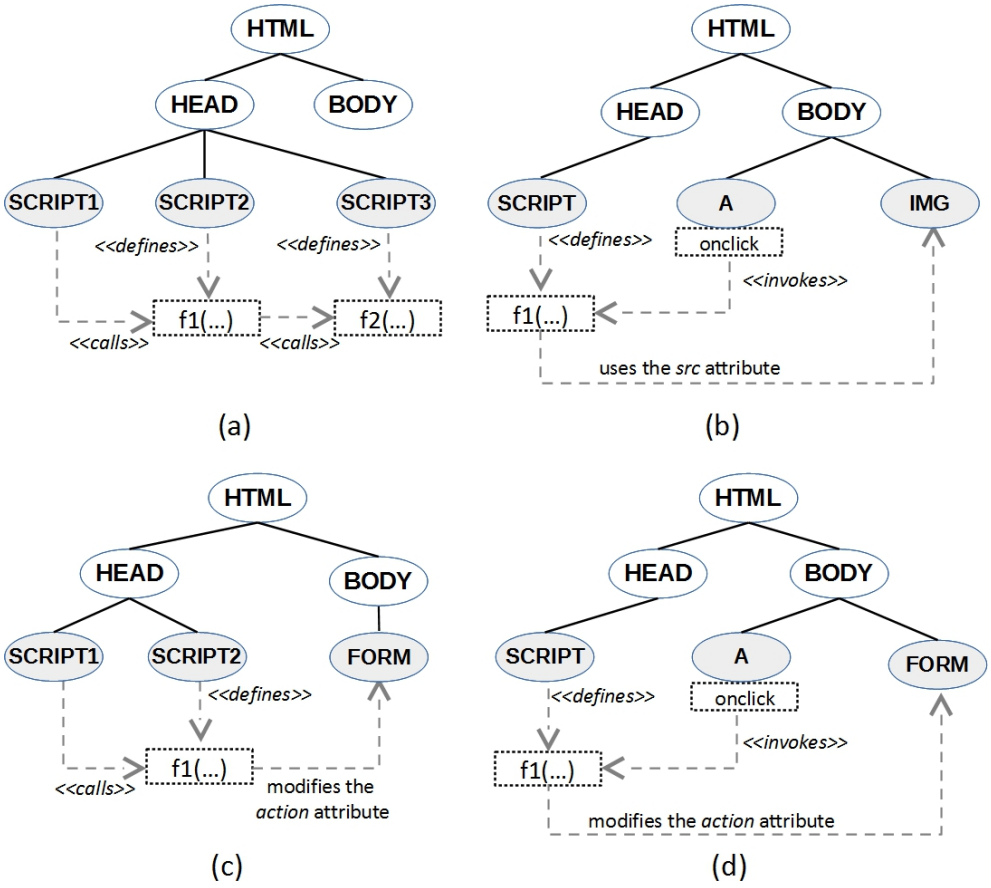


Figura 42. Ejemplo de dependencias transitivas.

El ejemplo de la Figura 42.c, muestra un fragmento del árbol DOM de la página donde el código del nodo *SCRIPT1* invoca a una función *f1*, definida en el nodo *SCRIPT2* (*SCRIPT1* → *SCRIPT2*), y el código de la función *f1* modifica el atributo *action* del nodo *FORM* (*FORM* → *SCRIPT2*).

Para el correcto procesamiento del nodo *FORM* (por ejemplo, para que el formulario se envíe de forma correcta), es necesario definir y ejecutar con anterioridad la función *f1* (esto implica que son necesarios los nodos *SCRIPT1* y *SCRIPT2*). Por lo tanto, ambos nodos son dependencias del elemento *FORM* (*FORM* → *SCRIPT1*).

Propiedad 4: Si *n1*→ $^{e|n}$ *n2* y *n3*→ *n2* porque *n2* es un nodo script que realiza una modificación en *n3*, entonces *n3*→ $^{e|n}$ *n1*.

El ejemplo de la Figura 42.d, muestra un fragmento del árbol DOM de una página en la que el manejador de evento *click* del nodo *A* invoca la función *f1*, definida en el nodo *SCRIPT* (*A* → $^{click|A}$ *SCRIPT*), y el código de la función *f1* modifica el atributo *action* del nodo *FORM* (*FORM* → *SCRIPT*).

Para que el nodo *FORM* sea procesado de forma correcta, cuando se emite el evento *click* sobre el nodo *A* (por ejemplo, para enviar el formulario), son necesarios tanto el nodo *SCRIPT* como el nodo *A*, es decir, ambos son dependencias del nodo *FORM* (*FORM* → $^{click|A}$ *A*).

3.1.2 Cálculo de los nodos y de los eventos automáticos relevantes

Uno de los objetivos principales de la fase de optimización es identificar el conjunto de nodos relevantes, todos ellos necesarios para la ejecución correcta de la secuencia de navegación.

Durante esta fase, el navegador opera de manera similar a cómo lo hace un navegador convencional:

1. En primer lugar, se inicia el proceso de carga de la página y de generación del árbol DOM.
2. Durante este proceso de carga, se descargan todos los elementos externos (por ejemplo, las hojas de estilo y los ficheros de script) a medida que aparecen durante el procesado del código fuente HTML.
3. Además, se ejecutan todos los nodos script de manera secuencial en el orden en el que van apareciendo en el documento HTML (la evaluación de estos scripts, puede dar lugar a nuevas descargas, nuevos scripts, etc.).
4. En último lugar, también se disparan los eventos que el navegador emite de manera automática, cuando se carga cada página de forma completa (por ejemplo, el evento *load* que se emite sobre el elemento *body*).

Con posterioridad, una vez que ha finalizado la construcción del documento HTML, el navegador reproducirá la secuencia de navegación, disparando los eventos sobre los elementos seleccionados, simulando así, la interacción del usuario con la página web (por ejemplo, realizando un clic sobre un elemento, moviendo el ratón sobre un nodo de la página, etc.), hasta que se inicia una navegación a una página nueva.

Durante todo este proceso, el componente de navegación monitoriza la ejecución de scripts para detectar las dependencias entre los nodos (en la validación experimental de este trabajo se ha utilizado la librería Mozilla Rhino, con las modificaciones necesarias para permitir este tipo de monitorización).

Para obtener todas las dependencias entre los nodos del árbol DOM de la página, se utilizan las reglas definidas en el apartado anterior.

Por ejemplo, cuando se ejecuta un nodo script, el componente de navegación interactúa con el motor de ejecución de JavaScript, para detectar qué funciones se invocan durante su ejecución, o a qué nodos del árbol DOM se accede durante este proceso de evaluación de JavaScript.

Si un nodo script utiliza alguna función definida en otro script del árbol DOM, de acuerdo con la primera regla de la Definición 1, el nodo que ha definido dichas funciones, se marca como dependencia del nodo script que las ha utilizado.

De forma análoga, si el código de un nodo script crea o modifica otro nodo, de acuerdo con la regla 3 de la Definición 1, este nodo script será una dependencia del nodo modificado.

Además, cuando se emite un evento (ya sea un evento automático o un evento generado por la secuencia de navegación), el componente de navegación realiza el siguiente análisis:

1. Por un lado, los nodos utilizados durante la ejecución de los manejadores asociados al evento.
2. Por otro lado, también analiza si se generan otros eventos y qué nodos se modifican durante la ejecución de éstos. Es decir, se generarán las dependencias apropiadas de acuerdo con las reglas de la Definición 2.

Una vez calculadas todas las dependencias, se construye el conjunto de nodos relevantes y para ello, se utilizan las siguientes reglas:

1. Los nodos que se utilizan en la secuencia de navegación de forma directa, son relevantes.

   Por ejemplo, si un paso de la secuencia de navegación genera un evento *click* sobre un nodo *A*, entonces ese nodo *A* es relevante.

2. Si un nodo *n* es relevante, todos sus ancestros son relevantes.

   Es necesario tener en cuenta que los ancestros podrían ser necesarios debido a las fases de *capture* y *bubbling* del modelo de eventos del árbol DOM.

3. Por definición, si el nodo *n1* es relevante y $n1 \rightarrow n2$ entonces *n2* es relevante (es decir, todas las dependencias de un nodo relevante son también relevantes).

4. Por definición, si un nodo *n1* es relevante, $n1 \rightarrow^{e/n} n2$, y el evento *e* fue disparado sobre el nodo *n*, entonces *n2* es relevante (es decir, si se emite el evento *e* sobre el nodo *n*, todas las dependencias condicionadas a que el evento *e* se dispare sobre *n* son también relevantes).

5. Con el objetivo de enviar de forma correcta los formularios, se aplican algunas reglas especiales sobre los nodos que intervienen en este proceso:

   a. Si un nodo formulario es relevante, también son relevantes todos los nodos de entrada (nodos *INPUT* o *TEXTAREA*) y de selección (nodos *SELECT*) que están contenidos en el formulario.

   b. Si un nodo *INPUT* o *SELECT* es relevante, el nodo formulario que lo contiene es relevante.

   c. Si un nodo selección es relevante, todos sus nodos hijos *OPTION* son relevantes.

6. Un pequeño conjunto de nodos, correspondientes a algunos tipos especiales de elementos, se consideran siempre como relevantes. Estos nodos son necesarios para procesar de forma adecuada, otros nodos del árbol DOM de la página.

Por ejemplo, el elemento *BASE* establece la URL base, lo que significa que las URLs especificadas por otros elementos son relativas al valor que indica dicho elemento.

A partir del conjunto de nodos relevantes, se puede calcular de manera sencilla el conjunto de nodos irrelevantes, que pueden ser eliminados durante la fase de ejecución:

1. En primer lugar, todos los nodos del árbol DOM que no están contenidos en el conjunto de nodos relevantes, se añaden al conjunto de nodos irrelevantes.

2. A continuación, se eliminan del conjunto de irrelevantes todos aquellos nodos que tienen un ancestro que también está contenido en dicho conjunto.

3. Como resultado, se obtienen los nodos raíz de los subárboles cuyos descendientes son todos irrelevantes. Dicho conjunto se denomina conjunto de subárboles irrelevantes.

4. En último lugar, para determinar cuáles de los eventos automáticos son necesarios para la correcta ejecución de la secuencia, el sistema comprueba, para cada evento automático, si alguno de los nodos relevantes tiene una dependencia derivada de ese evento (es decir, comprueba si algún nodo relevante, se ha visto afectado por la ejecución de alguno de los manejadores asociados al evento).

   En caso afirmativo, el evento se añade a la lista de eventos automáticos que deben emitirse en tiempo de ejecución, cuando se carga la página HTML.

La Figura 43, ilustra un ejemplo de cómo funciona el algoritmo que calcula los nodos irrelevantes.

En este ejemplo, los círculos de color gris, representan a los nodos identificados como relevantes, mientras que los círculos de color blanco, representan a los nodos identificados como irrelevantes:

- En primer lugar, se añaden a la lista de nodos irrelevantes, todos los nodos marcados en color blanco: *LINK*, *UL*, *LI* (ambos nodos), *DIV* y *SPAN*.

- A continuación, se eliminan de este conjunto, aquellos nodos que tienen un ancestro que también es irrelevante:

  - Se eliminan los nodos *LI*, porque su antecesor *UL* es irrelevante.

  - Se elimina el nodo *SPAN*, porque su antecesor *DIV* es irrelevante.

- El conjunto final de nodos irrelevantes, contendrá sólo aquellos nodos que son raíz de un subárbol irrelevante: *LINK*, *UL*, y *DIV*.

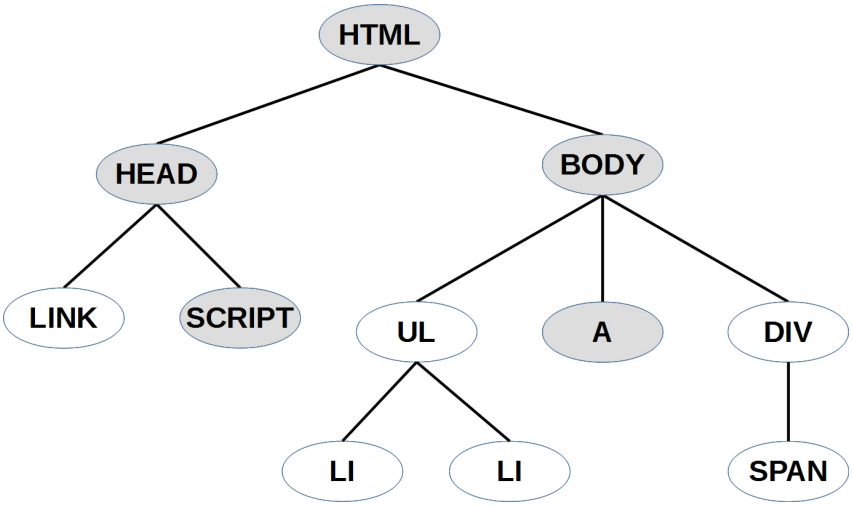


Figura 43. Ejemplo de cálculo de nodos irrelevantes.

Almacenar sólo las raíces de los subárboles irrelevantes, permite:

a) Por un lado, minimizar la cantidad de información guardada de forma persistente, para identificar las partes irrelevantes de un documento HTML.

b) Por otro lado, optimizar el proceso de identificación de fragmentos irrelevantes. Cada vez que se localiza un nodo irrelevante dentro de un documento HTML, se podrá descartar de manera automática, todo el subárbol que tiene como raíz a ese nodo irrelevante.

A continuación, en la Figura 44, se detalla un ejemplo completo, que ilustra cómo funciona el mecanismo para calcular los nodos relevantes, durante la ejecución de la secuencia de navegación web.

Esta figura, muestra un fragmento de un árbol DOM, sobre el que se ejecuta una secuencia, que emite un evento *click* sobre el nodo de tipo *A*.

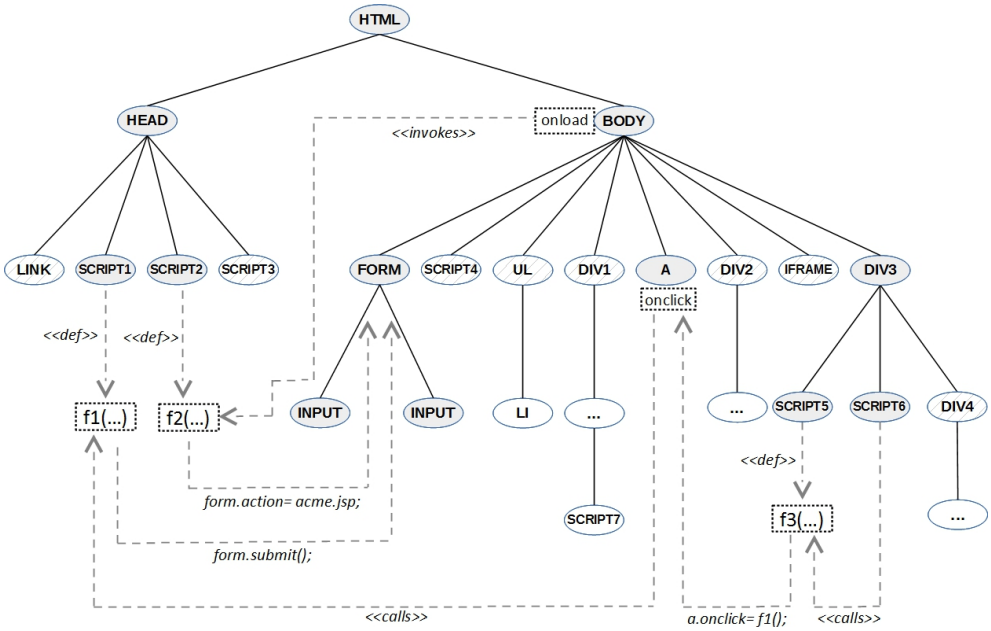


Figura 44. Ejemplo completo del proceso de identificación de nodos relevantes.

Los nodos relevantes, se muestran en color gris y han sido calculados de la siguiente manera:

1. De acuerdo con la regla 1, el nodo *A* es relevante (porque es el objetivo de la acción).

2. De acuerdo con la regla 2, todos los ancestros de *A* son relevantes: *BODY* y *HTML*.

3. De acuerdo con la regla 3, todas las dependencias de *A* son relevantes: *SCRIPT5* y *SCRIPT6* (y también sus ancestros: *DIV3*).

   En este caso, son necesarios porque ejecutan el código script que modifica el manejador del evento *click* del nodo *A*, cuando se carga la página.

   a) La función *f3* (definida en *SCRIPT5*), establece el manejador del evento *click* del nodo *A*, por tanto *A* → *SCRIPT5*.

   b) *SCRIPT6*, que se ejecuta cuando se carga la página, invoca a la función *f3*, por tanto *SCRIPT6* → *SCRIPT5*, y aplicando las propiedades de transitividad, *A* → *SCRIPT6*.

4. De acuerdo con la regla 4, también son relevantes todas las dependencias de *A*, condicionadas a que el evento *click* se dispare sobre el nodo *A*: *SCRIPT1* y *FORM* (y todos sus ancestros: *HEAD*).

   Dichos nodos, son necesarios porque el manejador del evento *click* del nodo *A* invoca la función *f1*, definida en *SCRIPT1*, y esta función es la que envía el formulario *FORM*.

   a) El manejador del evento *click* del nodo *A* invoca la función *f1* definida en *SCRIPT1*, por tanto $A \rightarrow^{click|A} SCRIPT1$.

b) La función *f1* utiliza el nodo *FORM*, por tanto *SCRIPT1* → *FORM*, y aplicando las reglas de transitividad, *A* → $^{click|A}$ *FORM*.

5. De acuerdo con la regla 5, si un nodo *FORM* es relevante, también lo son todos los nodos *INPUT* contenidos en el formulario: *INPUT1* e *INPUT2*.

   Para enviar de forma correcta el formulario, son necesarios todos sus campos de entrada.

6. De acuerdo con la regla 3, todas las dependencias del nodo *FORM* son relevantes: *SCRIPT2* y *BODY* (y todos sus ancestros, que ya están incluidos en el conjunto de nodos relevantes).

   Dichos elementos, son necesarios porque el manejador del evento *load* del nodo *BODY* invoca una función definida en *SCRIPT2* y esta función modifica el atributo *action* del nodo *FORM*.

   a) El manejador de evento *onload* del nodo BODY, invoca la función *f2* definida en *SCRIPT2*, por lo tanto *BODY* →$^{load|body}$ *SCRIPT2*.

   b) La función *f2* (definida en *SCRIPT2*) modifica el atributo *action* del nodo *FORM*, por lo tanto *FORM* → *SCRIPT2*, y tras aplicar las reglas de transitividad, *FORM* →$^{load|body}$ *BODY*.

Los nodos que se muestran en la Figura 44 con líneas de trazos, son los que han sido identificados como raíces de subárboles irrelevantes, y por lo tanto, pueden ser descartados en posteriores ejecuciones de la secuencia.

Por ejemplo, el nodo *LI*, no es una raíz de un subárbol irrelevante porque lo es su ancestro, el nodo *UL*. Si durante el proceso de construcción del árbol DOM, se descarta el subárbol que tiene como raíz al nodo *UL*, también se descartará el nodo *LI*.

El evento automático *load*, emitido sobre el nodo *BODY*, debe ser añadido a la lista de eventos automáticos necesarios, porque el nodo *FORM*, que es un nodo relevante, tiene una dependencia derivada de él (*FORM* → $^{load|body}$ *BODY*).

Esto se debe a que, para enviar el formulario de forma correcta, el manejador del evento *load* del elemento BODY (*onload*) invoca a la función *f2* y esta función establece el atributo *action* del formulario (el atributo *action* de un formulario establece la URL de envío de ese formulario).

3.1.3 Identificación de los subárboles relevantes en la fase de optimización

Una vez que se han calculado los nodos que identifican a las raíces de los subárboles irrelevantes, es necesario generar expresiones para identificar a cada uno de estos subárboles durante la fase de ejecución.

Existen dos requisitos para este proceso, que se detallan a continuación:

1. Por un lado, las expresiones generadas deberían ser resistentes a pequeños cambios en la página, porque en los sitios web reales, por lo general, hay

pequeñas diferencias entre el árbol DOM de la misma página cargada en diferentes momentos (por ejemplo, puede aparecer un nuevo anuncio de publicidad o se pueden mostrar registros de datos diferentes).

2. Por otro lado, la comprobación de si un nodo encaja con una expresión, debe ser un proceso muy eficiente, porque durante la fase de ejecución, se comprueba si cada uno de los nodos presentes en el código HTML, encaja con alguna de esas expresiones que identifican a cada uno de los fragmentos irrelevantes.

Para identificar de forma unívoca a un nodo en el árbol DOM, se utiliza una expresión XPath [XPATH].

Cada una de estas expresiones, debe identificar a un único nodo en todo el árbol, y para ello, puede contener, tanto información de ese nodo, como información de alguno de sus ancestros.

Además, estas expresiones no deberían ser demasiado específicas, como para que se vean afectadas por pequeños cambios en el código fuente.

Por ese motivo, se utiliza como base el algoritmo descrito en [MPBL11], modificado y mejorado para alcanzar estos objetivos.

La idea fundamental de este algoritmo consiste en construir una expresión, con el mínimo número de nodos posible, ya que ello contribuirá a aumentar la resistencia a cambios.

Para identificar a cada uno de estos nodos, que forman parte de una expresión, se podrán utilizar los siguientes elementos:

1. El nombre de la etiqueta HTML de los elementos (por ejemplo *BODY*, *FORM*, etc.).
2. El nombre y el valor de los atributos del elemento.
3. El texto contenido en el interior de un nodo hoja.
4. En algunos casos, también será necesario utilizar la posición que ocupa un nodo, tomando como referencia el nodo padre y considerando todos los elementos hermanos (al mismo nivel), con la misma etiqueta.

   Por ejemplo, dado un nodo con etiqueta *DIV*, con dos hijos directos con etiqueta *SPAN*, utilizando este método de identificación, el primero de ellos se identificará con *//DIV/SPAN[1]*, y el segundo con *//DIV/SPAN[2]*.

Un concepto importante para la generación de estas expresiones, es lo que en este trabajo se denomina “expresión de nodo” que contendrá información para identificar a un solo nodo, dentro del árbol DOM. Estas expresiones tendrán siempre el siguiente formato:

//TagName[@a1=”v1” and ... and @am=”vm” and text()=”t” or position]

- *TagName*: representa la etiqueta de un nodo.

- a_i, $v_{i\ i=\{1,\ldots,m\}}$: representan los nombres y valores de los atributos de un nodo.
- *text()*: representa el texto de un nodo hoja.
- *position*: hace referencia a la posición que ocupa un nodo con respecto a los nodos hermanos, con la misma etiqueta (por ejemplo, en el caso de representar filas de tablas, la posición 2 le correspondería al segundo hijo directo de un nodo *TABLE* que tuviese etiqueta *TR*).

En una expresión de nodo, el único componente que es obligatorio es la etiqueta.

Si un nodo se puede identificar de forma unívoca, utilizando su expresión XPath de nodo, el proceso de generación de la expresión se detiene y la salida resultante será esa expresión.

En caso contrario (esto sucede cuando más de un nodo del árbol DOM se representa con la misma expresión de nodo), se busca un nodo que sí pueda ser representado de forma unívoca, utilizando su expresión de nodo, en el camino que va desde el nodo que se está procesando, hasta la raíz del árbol, pasando por todos los ancestros.

Una vez que se encuentra ese nodo, la expresión para identificarlo se concatena a la expresión final y a continuación, el algoritmo se aplica de nuevo sobre el nodo objetivo, pero esta vez, considerando sólo el subárbol que tiene como raíz el nodo que se ha identificado de forma unívoca.

De esta manera, el resultado del algoritmo será una expresión XPath que contendrá la representación de varios nodos del árbol DOM de la página:

$$//x_1//x_2//..//x_n$$

Y donde $//x_{i\ i\in\{1,\ldots,n\}}$ son las expresiones de nodo, que identifican de forma única, a un nodo dentro del subárbol, considerado en cada una de las iteraciones del algoritmo.

Se define la longitud de una expresión, como el número de nodos que han sido necesarios para generar la expresión final.

```
Algorithm: Generate an XPath-like expression to identify a node in a DOM tree
      * X = GenerateExpression(n,T)
Inputs:
      * n, the target node to be identified by the expression
      * T, the DOM tree where n is contained
Output:
      result, the XPath-like expression to uniquely identify n in T.

 1: result = "";      # Initialize the variable that will contain the result expression
 2: S = T;            # Initialize the variable that will contain the subtree considered in each iteration
 3: m = null;         # Auxiliary variable that will contain the node analyzed in each iteration
 4:
 5: Repeat {          # Iterate until the target node n can be uniquely identified in S
 6:    m = n;         # Initialize m to the target node n
 7:    x = null;      # Initialize the variable that will contain the node expression generated to identify m
 8:
 9:    # Iterate from n to the root of S until a node which can be uniquely identified is found or the root
10:    # of S is reached
11:    While (x==null && m!= root(S)) {
12:       x = getNodeExp(m, S);   # Returns an expression to uniquely identify m in S
13:                               # or null if such expression cannot be generated
14:       If (x != null) {        # The node can be uniquely identified in S
15:          result = result + x;
16:          S = <the subtree whose root is m>;
17:       } else {                # The node cannot be uniquely identified in S
18:          m = parent(m,T);     # Analyze the parent node in the tree
20:       }
21:    }
22:    If (m=root(S)) {        # No node can be uniquely identified in the path from n to the root of S.
23:       Let m' be the child of m which is in the path to the target node n;
24:       x =  getChildNodeExp(m',S);     # Returns an expression to uniquely identify
25:                                       # m' as a child of m in S, using the node position if necessary
26:       result = result + x;
27:       S = <the subtree whose root is m'>;
28:    }
29: } Until (m==n);
30: return result;
```

Figura 45. Algoritmo para la generación de expresiones estilo XPath.

La Figura 45, muestra el algoritmo completo para la generación de las expresiones XPath, que identifican de forma única, a un nodo *n* contenido dentro del árbol DOM *T* de una página web.

En este algoritmo, el bucle principal itera hasta que el nodo *n* puede ser identificado de manera unívoca dentro del subárbol seleccionado en cada iteración (en la primera iteración se considera todo el árbol DOM).

El bucle *while* itera desde el nodo *n* hasta la raíz del subárbol, hasta que encuentra un nodo que puede ser identificado de manera unívoca, dentro de ese subárbol. Cuando se encuentra ese nodo, se genera una expresión de nodo que lo identifica (variable *x*) y esa expresión se añade al resultado final (variable *result*).

En la siguiente iteración del bucle, este nodo que se ha identificado de manera única a través de una expresión de nodo, representará la raíz del subárbol.

La función *getNodeExp*, recibe como entrada un nodo y un subárbol e intenta generar una expresión XPath de nodo que identifica de manera única a ese nodo dentro del subárbol.

Si esa expresión no puede ser generada, la función devuelve *null*.

Al final del bucle, se considera un caso especial, para tratar aquellas situaciones en las que ningún elemento desde el nodo inicial *n* hasta la raíz del subárbol, se puede identificar de manera única con una expresión de nodo.

En esos casos, se utiliza la función *getChildNodeExp*, sobre el nodo hijo de la raíz del subárbol *S* (raíz del árbol actual), que se encuentra en el camino desde *n* a *S*. Esta función actúa de manera similar a *getNodeExp*, con las siguientes diferencias:

1. En ningún caso devuelve *null*.

   Siempre se aplican las consideraciones detalladas en los siguientes puntos.

2. La expresión generada empieza por "/" en vez de "//".

   Esto significa, que el nodo debe ser un hijo directo del último nodo cuya expresión de nodo se ha añadido a la expresión resultado (por ejemplo, cuando la expresión de nodo que identifica a *S* ha sido añadida a la variable *result* en la iteración anterior).

   Esto permite diferenciar a este nodo, de otros nodos que puedan encajar con la misma expresión de nodo, y no sean hijos directos del nodo raíz de *S*.

3. Si es necesario, también se puede utilizar la posición para distinguir de manera única a dos nodos, descendientes directos del nodo raíz de *S*.

De esta manera, la expresión XPath tendrá el siguiente formato:

$$\textbf{//}x_1\ [\text{“//”} \mid \text{“/”}]x_2 \ldots [\text{“//”} \mid \text{“/”}]x_n$$

Cabe destacar que todas las expresiones comenzarán siempre con "//", porque en caso de que no aparezca antes ningún nodo que se pueda identificar de manera única durante la primera iteración, los nodos HTML, BODY y HEAD siempre se podrán identificar de manera única, utilizando el nombre de su etiqueta (los documentos HTML sólo tienen una cabecera, un cuerpo y un nodo raíz; y si el código fuente contuviese varias etiquetas de estos tipos, sería necesario realizar un proceso de normalización del documento para poder construir su árbol DOM).

A continuación, se detalla el funcionamiento de la función *getNodeExp*. Esta función, trata de generar una expresión que pueda identificar, de manera única, a un nodo dentro de un subárbol. Esta función sólo utiliza el nombre de la etiqueta del elemento, sus atributos y su texto asociado en el caso de tratarse de un nodo hoja.

En primer lugar, esta función trata de identificar al nodo utilizando sólo su etiqueta. Si con esto no es suficiente (cuando existen varios nodos en el subárbol con la misma etiqueta), entonces utiliza también los atributos.

El algoritmo considera a algunos atributos más relevantes que otros.

Por ejemplo, el atributo *id*, es muy probable que sirva por sí solo para identificar a un nodo dentro del árbol DOM de la página, ya que se utiliza para asignar

identificadores únicos a los nodos. Otros atributos considerados como más relevantes son: *name*, *title*, *alt*, *value*, *for*, *src*, *action*, *href*, *class*, etc.

Además, el algoritmo también considera a otros atributos como menos relevantes, a la hora de identificar a un nodo. Se trata, sobre todo, de atributos que representan a valores numéricos, más susceptibles a cambios y, por lo tanto, su utilización daría lugar a expresiones más débiles, con menor resistencia a cambios. Ejemplos de estos atributos son: *cellpading*, *cellspacing*, *type*, *method*, *content*, *width*, *height*, *align*, *rel*, etc.

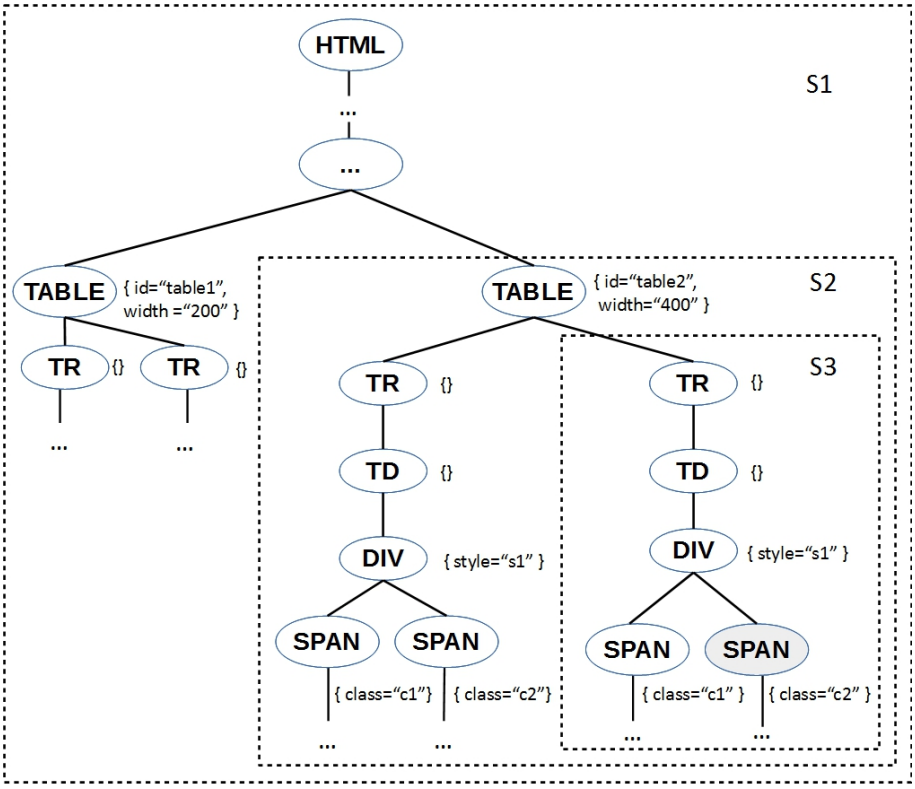
En primer lugar, el algoritmo trata de utilizar sólo los atributos considerados más relevantes.

Si con esto no es suficiente, se incluyen todos los atributos excepto los considerados menos relevantes. Por último, si fuese necesario, se incluyen también los atributos menos relevantes.

Si con los atributos no es posible obtener la expresión que identifique al nodo de forma única, el algoritmo utiliza también el texto de este nodo, en caso de tratarse de un nodo hoja.

Para ello, en primer lugar, intenta utilizar sólo el nombre de la etiqueta y el texto y si no fuese suficiente, utilizaría también los atributos, de la misma forma que se ha detallado con anterioridad.

La siguiente figura, ilustra un ejemplo de generación de una expresión XPATH, que identifica a un nodo (con etiqueta *SPAN*) dentro de un árbol DOM de una página HMTL.



First iteration:
S = S1
x = //TABLE[@id="table2"]
result = //TABLE[@id="table2"]

Second iteration:
S = S2
x = /TR[2]
result = //TABLE[@id="table2"]/TR[2]

Third iteration:
S = S3
x = //SPAN[class="c2"]
result = //TABLE[@id="table2"]/TR[2]//SPAN[class="c2"]

Figura 46. Ejemplo de generación de expresión XPath.

El elemento a identificar será el nodo con etiqueta *SPAN* (el que tiene un color más oscuro en la figura). En la primera iteración del algoritmo, se considera todo el árbol DOM de la página HTML (*S1*).

En esta primera iteración, no es posible identificar de forma única al nodo SPAN con una expresión de nodo XPATH, al existir otro nodo dentro de la página HTML con la misma etiqueta y con los mismos atributos (en este caso, el atributo *class* con valor *c2*).

A continuación, el algoritmo busca otro nodo que se pueda identificar de forma única con una expresión de nodo, siguiendo un camino que va desde el nodo *SPAN* hasta la raíz del documento, pasando por todos los ancestros del nodo *SPAN*.

El primer elemento que cumple con estas características, es el nodo con etiqueta *TABLE* y con atributo *id* con valor *table2*.

Este nodo se puede identificar de manera unívoca porque el valor del atributo *id* es único dentro de toda la página HTML (cabe destacar que el nodo *TABLE* también se puede identificar con el atributo *width*, pero éste se descarta al ser considerado un atributo poco relevante).

En la segunda iteración (*S2*), el algoritmo considera el subárbol que va desde el nodo objetivo *SPAN* hasta el nodo *TABLE*, identificado de manera única en la primera iteración.

En el camino que va entre estos dos nodos, no existe ningún nodo que se pueda identificar de manera única con una expresión de nodo (nodos *SPAN*, *DIV*, *TD* y *TR*).

Por lo tanto, para generar la expresión, el algoritmo utiliza el nodo *TR* que está en el camino entre el nodo *SPAN* y el nodo *TABLE*. La expresión generada para identificar a este nodo, comenzará con "/" para indicar que se trata de un descendiente directo del nodo *TABLE*.

Además, es necesario utilizar la posición de este nodo, dado que el nodo *TABLE*, contiene dos nodos *TR* que son exactamente iguales.

En la tercera iteración (*S3*), el algoritmo considera el subárbol que va desde el nodo objetivo *SPAN* hasta el nodo *TR*, identificado en la segunda iteración.

En esta iteración, el nodo SPAN sí que puede ser identificado de manera única en el subárbol seleccionado (utilizando el nombre de su etiqueta, junto con el atributo *class*, dado que en el subárbol existe otro nodo con etiqueta *SPAN*).

Tras finalizar la tercera iteración, se ha obtenido una expresión que identifica de manera única al nodo SPAN dentro de todo el árbol DOM, y esta expresión estará compuesta por los tres fragmentos obtenidos en cada una de las iteraciones.

//TABLE[@id="table2"]/TR[2]//SPAN[class="c2"]

3.1.4 Identificación de documentos en los que se realizará la búsqueda de fragmentos irrelevantes

Una vez que se ha generado la información con los fragmentos irrelevantes de una página HTML, es necesario identificar a todos los documentos en los que se podrá utilizar dicha información, para construir un árbol DOM minimizado.

El objetivo de este proceso es el de identificar todas las direcciones de Internet, en las que pueden aparecer los nodos irrelevantes que se han encontrado en una página determinada.

Para ello, es necesario obtener la siguiente información sobre la petición HTTP, que se ha realizado para alcanzar cada una de las páginas:

1. Método HTTP de acceso (GET o POST).

2. URL de acceso. Esto incluye el protocolo (HTTP o HTTPS), el nombre del host, el puerto, la ruta completa y los parámetros.
3. En el caso de que el método sea POST, también se extraerán los parámetros incluidos en el cuerpo de la petición.

Una vez se ha extraído esta información, se utilizarán los siguientes elementos para identificar a los documentos en los que se intentará localizar el conjunto de nodos irrelevantes:

1. Método de acceso (GET o POST).
2. URL de acceso simplificada. Esto incluye el protocolo, el nombre de la máquina y el puerto y la ruta completa, pero sin incluir los parámetros de la URL.
3. Nombres de los parámetros utilizados en la petición (tanto los parámetros incluidos en la URL de la petición GET, como los parámetros incluidos en el cuerpo del mensaje en una petición POST).

   Es necesario subrayar que se utilizarán sólo los nombres de los parámetros, ignorándose los valores que éstos puedan tener.

   Utilizar sólo los nombres de los parámetros, permite que un mismo conjunto de fragmentos irrelevantes, se pueda utilizar con muchas direcciones diferentes.

A continuación, se muestra un pequeño ejemplo.

La Figura 47, muestra una dirección de Internet que se ha alcanzado a través de un formulario que tiene dos parámetros llamados "*q"* y "*filter*".

El envío de este formulario se realiza a la página "*/search/search.html*", a través de una petición HTTP GET.

Si en el documento alcanzado a través de esta petición, se han encontrado fragmentos irrelevantes identificados mediante expresiones XPath, estos fragmentos se podrán utilizar en todos los documentos que cumplan los siguientes requisitos.

1. El documento se alcanza a través de una petición GET.
2. La petición se realiza a la URL *http://www.acme.com/search/search.html*.
3. La petición incluye los parámetros "*q*" y "*filter*" (para cualquier valor que éstos puedan tener) y ningún otro parámetro adicional.

De este modo, los mismos fragmentos irrelevantes se podrán utilizar en distintas búsquedas, realizadas sobre el mismo formulario.

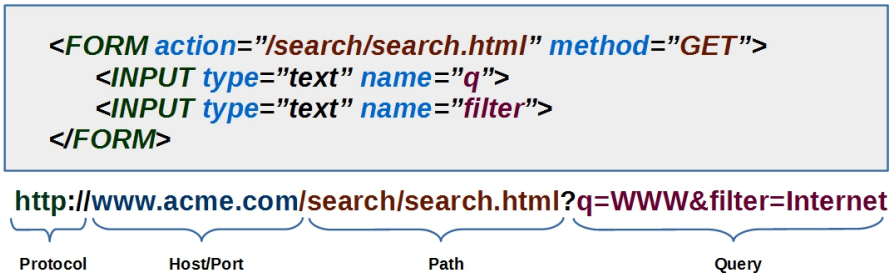


Figura 47. Ejemplo de obtención de información de una URL.

Por ejemplo, si los fragmentos irrelevantes han sido localizados en la dirección *http://acme.com/search/search.html?q=WWW&filter=Internet*, ese mismo conjunto de fragmentos, se podrá utilizar para construir un árbol DOM minimizado en la dirección *http://acme.com/search/search.html?q=WebServices&filter=REST*.

En algunas ocasiones, los servidores web incluyen los identificadores de sesión en las URLs de acceso. En estos casos, la URL para acceder a un mismo documento varía con cada nueva sesión generada en el servidor.

Para dar soporte a estos escenarios, el proceso de identificación de documentos tiene en cuenta algunos mecanismos estándar de generación de identificadores de sesión (por ejemplo, el mecanismo que utilizan los servidores web desarrollados utilizando tecnologías Java2EE).

Cuando se detecta un patrón conocido de identificador de sesión dentro de una URL, se utilizará una expresión regular que permitirá identificar a todas las posibles URLs, sin tener en cuenta el identificador de sesión.

La Figura 48, ilustra un ejemplo de dos páginas web similares, que han sido alcanzadas ejecutando la misma secuencia de navegación, modificando los parámetros de entrada de esta secuencia, para establecer palabras clave de búsqueda diferentes.

En el primer caso, la secuencia ha realizado la búsqueda por palabra clave "*Java*" y en el segundo caso, ha realizado la búsqueda por palabra clave "*Web Service*".

En ambas búsquedas, la estructura de las páginas alcanzadas es la misma, ya que las dos páginas se habrán generado utilizando la misma *plantilla*, y la parte que varía, es la que muestra los datos concretos de cada búsqueda.

Si se ejecutase la fase de optimización para identificar los fragmentos irrelevantes sobre la primera de ellas, el conjunto de nodos irrelevantes identificado durante este proceso, se podrá utilizar también sobre cualquier otra página alcanzada ejecutando la misma secuencia con otros parámetros (o lo que es lo mismo, cualquier otra página generada con la misma *plantilla*).

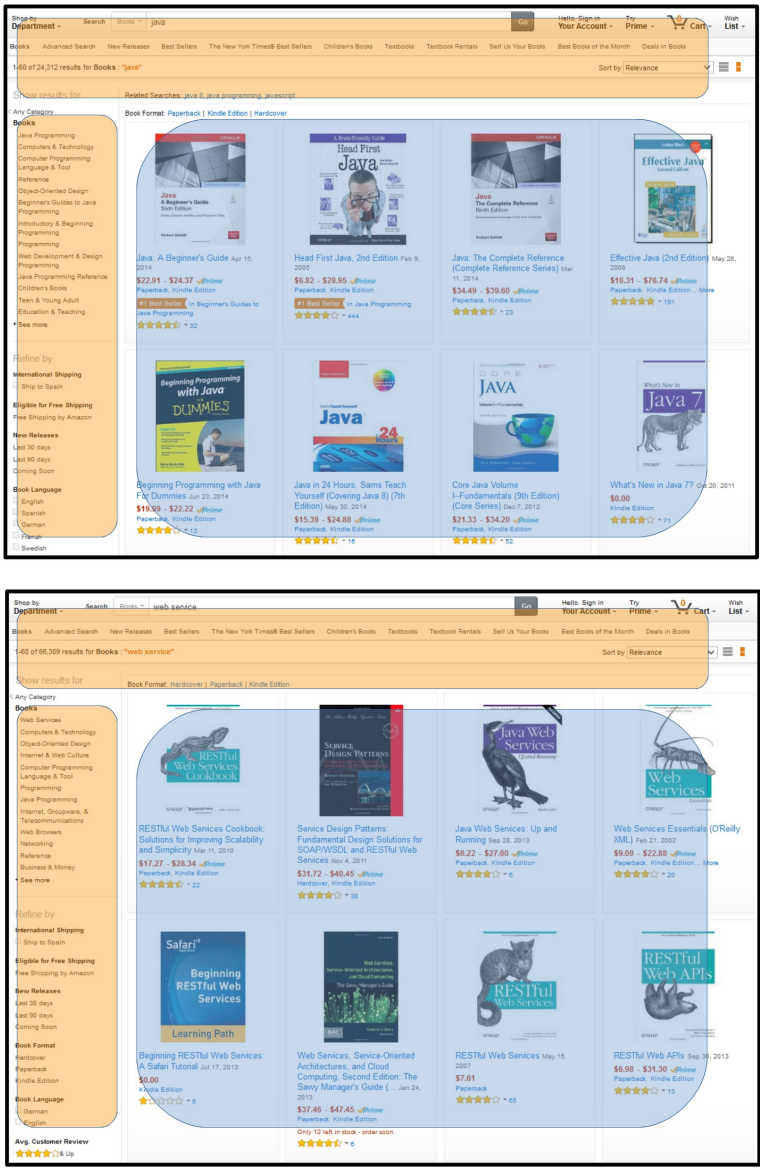


Figura 48. Ejemplo de documentos similares en búsquedas diferentes.

3.1.5 Fase de ejecución

Durante la fase de optimización, se han generado los fragmentos irrelevantes dentro del árbol DOM de la página HTML, y se han generado expresiones XPATH que permitirán identificarlos, con posterioridad, de forma única.

A continuación, durante la siguiente fase, se utilizará esta información para construir un árbol DOM reducido, descartando estos fragmentos irrelevantes.

Ésta es la llamada fase de ejecución, e involucra todas las ejecuciones posteriores de la misma secuencia de navegación.

El funcionamiento del componente de navegación a medida durante la fase de ejecución, se puede resumir de la siguiente manera: antes de cargar cada una de las páginas HTML, el componente de navegación comprueba si existe información

de optimización asociada a esa página (conjunto de expresiones XPath para identificar a los fragmentos irrelevantes).

Para determinar si existen fragmentos asociados a una página web, se utiliza el mecanismo de identificación de documentos detallado en el apartado 3.1.4.

En caso afirmativo, esa información se utiliza para construir un árbol DOM reducido descartando la mayor cantidad de fragmentos posible. Por otro lado, también se comprueba si existe información sobre los eventos automáticos que deben ser disparados en esa página.

El proceso de comprobación de si un nodo se corresponde con un fragmento irrelevante debe ser muy eficiente. Este chequeo, se ejecutará para un número importante de los nodos presentes en el código fuente HTML, con el objetivo de decidir si cada uno de estos nodos, debe ser añadido o no al árbol DOM final de la página.

Por este motivo, no se utiliza un algoritmo convencional de emparejado XPath.

En vez de esto, se ha diseñado un algoritmo de comprobación basado en que las expresiones XPath generadas, van a seguir un formato predeterminado, y todas ellas van a cumplir siempre ciertas restricciones (el sistema no permite cualquier tipo de expresión XPATH para representar a los nodos irrelevantes).

Esta peculiaridad, permitirá un proceso de comprobación mucho más rápido, durante la generación del árbol DOM de la página.

A continuación, se detalla el algoritmo que comprueba si un nodo se encuentra dentro del conjunto de fragmentos irrelevantes, representados mediante expresiones XPath.

```
Algorithm: Check if a node matches with any of the XPath-like expressions which identifies the root nodes of
           the irrelevant subtrees
    * result = CheckIfIrrelevantNode(n,X)
Inputs:
    * n, the target node to check if it matches with any expression.
    * X={X1, …Xr}, where each Xi i∈{1,...,r} is an XPath-like expression identifying the root node of an irrelevant
      subtree. Each Xi is an expression with the following format: //xi1 ["//" | "/"]xi2 … ["//" | "/"]xit where ["//"
      | "/"]xik k∈{1,...,t} is a node expression to identify a node using its tag name, attributes and/or text.
    * T, the DOM tree built up to the moment, and where the node n will be added if it does not match with
      any expression in X.
Output:
    True if n matches with any Xi i∈{1,...,r} or false in other case.

 1: i = 1;      # Auxiliary expression counter
 2: while (i<=r) {          # Process one XPath-like expression in each iteration
 3:    m = n;               # Initialize m to the target node n
 4:    k = length(Xi);      # Auxiliary counter, initialized to the number of node expressions in Xi
 5:    If (matches(m,xik) {    # If the target node matches with the last node expression
 6:       m = parent(m);    # Take the parent node
 7:       k = k - 1;        # Point to the previous node expression
 8:       While (k>0 && m!=null) {      # While there are node expressions left and parent nodes to match
 9:          p = xik;                   # Partial expression initially set to the current node expression
10:          while (xik is preceded by "/") {  # Add to p all the consecutive previous node expressions
11:             k = k - 1;                     # concatenated by "/"
12:             p = xik + "/" + p;
13:          }
14:          matched = false;
15:          while (m != null && !matched) {   # Iterates over nodes in the path to the tree root
16:             N = [m];                       # Node list, initially containing the current node
17:             m' = m;
18:             Repeat (length(p) -1) times {  # Add to N the same number
19:                m' = parent(m',T);          # of nodes as node expressions are in p
20:                append(N,m');
21:             }
22:             if (matches(N,p) {      # If the partial expression matches with the list of nodes
23:                matched = true;
24:                m = parent(m',T);    # Continue with the parent node of the ones matched in this iteration
25:                k = k - 1;           # and the node expression previous to the ones matched in this iteration
26:             } else {
27:                m = parent(m,T);     # Try to match p from the parent node of the current one
28:             }
29:          }
30:       }
31:       If (k=0) {    # All the node expressions of Xi have been matched
32:          return true;
33:       } else {
34:          i = i+1;   # Analyze the next XPath-like expression
35:       }
36:    } else {
37:       i = i+1;      # Analyze the next XPath-like expression
38:    }
39: }
40: return false;
```

Figura 49. Algoritmo para buscar nodos no relevantes.

El algoritmo, recibe como entrada el nodo *n*, el cual se quiere determinar si es o no irrelevante, junto con el conjunto de expresiones XPath, y comprueba si alguna de las expresiones XPath, encaja con el nodo *n*.

Si alguna expresión identifica al nodo actual, ese nodo se considerará irrelevante, y será descartado del DOM final de la página (junto con todo el subárbol que está por debajo de él).

Para ello, el bucle más externo itera sobre las expresiones XPath generadas en la fase de optimización.

El algoritmo comprueba si hay nodos en el camino que va desde el nodo *n*, hasta la raíz del árbol DOM, que encajen con cada uno de los fragmentos individuales que forman cada una de estas expresiones XPath.

Para comprobar si el nodo encaja con una expresión XPath, la primera condición que ha de cumplirse es que el último fragmento de la expresión XPath, debe identificar al nodo que se está procesando.

En caso de que esta condición no se cumpla, el algoritmo considera que el nodo no encaja con esa expresión.

En caso contrario, el algoritmo comprueba si hay más fragmentos en la expresión XPath, y estos fragmentos deben ser emparejados, por orden, con nodos que se encuentren en el camino desde el nodo objetivo hasta la raíz del árbol.

Esta comprobación se lleva a cabo en el segundo de los bucles, el cual itera sobre los fragmentos restantes de la expresión XPath.

Cada una de las iteraciones del bucle más interno, va subiendo en el árbol, hasta encontrar un nodo que se corresponda con el fragmento de expresión actual.

Además, debe considerar los casos especiales en los que la expresión comienza con "/", en vez de "//" (cuando se trata de un hijo directo).

Para ello, el tercer bucle concatena todos los fragmentos consecutivos que empiezan por "/", para generar una sub-expresión parcial con ellos.

A continuación, el cuarto bucle itera sobre los nodos que se encuentran en el camino hacia la raíz, intentando localizar una sucesión de nodos consecutivos, que se correspondan con la sub-expresión parcial.

Cabe destacar que, en el segundo bucle, cuando la expresión no empieza con "/", la sub-expresión parcial sólo contendrá un elemento y el cuarto bucle sólo intentará localizar un nodo.

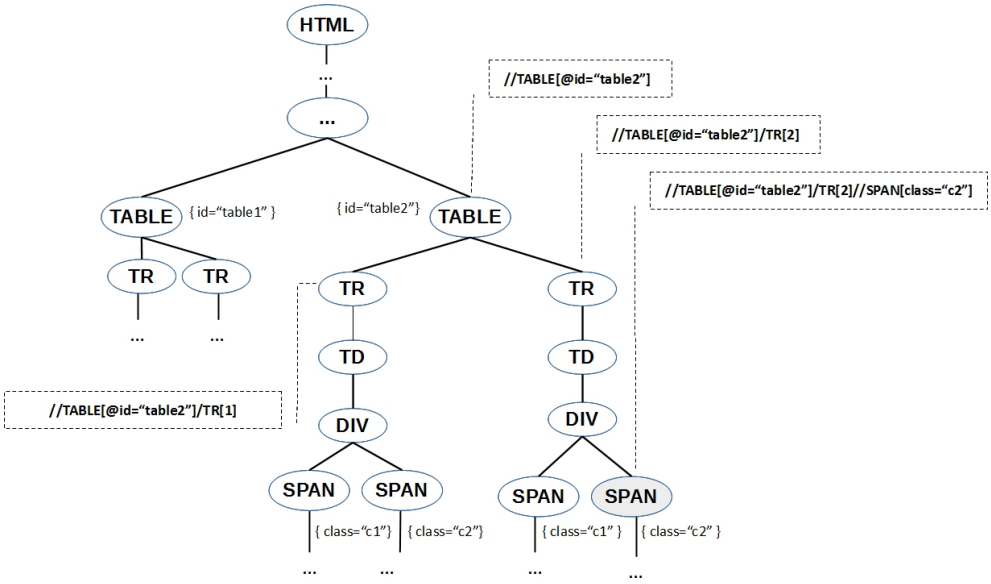


Figura 50. Ejemplo de emparejamiento de nodos irrelevantes.

A continuación, en la Figura 50, se detalla cómo se construiría el árbol DOM del ejemplo de la Figura 46, aplicando este algoritmo de emparejamiento de nodos irrelevantes y considerando la expresión XPath que se generó para descartar el nodo *SPAN* sombreado en la figura:

1. Todos los nodos que no sean de tipo *SPAN*, o que siendo de tipo SPAN, no incluyan un atributo *class*, con valor *c2*, serán descartados sin más procesamiento, y se añadirían al árbol DOM, ya que no encajan con la expresión.

2. Cuando aparece el primer nodo *SPAN* con un atributo *class*, con valor *c2* (se trata del segundo nodo SPAN, de los cuatro que contiene el documento), se construye la expresión parcial *//TABLE[@id="table2"]/TR[2]* para identificar a un nodo *TABLE*, con *id table2* y a su segundo descendiente directo, con etiqueta *TR*.

   Los nodos *TR* y *TABLE*, ascendientes de este nodo *SPAN*, no encajan con esta sub-expresión, dado que el nodo *TR* es el primer descendiente directo del nodo *TABLE* con etiqueta *TR* y no el segundo, que es lo que indica la expresión.

   Por lo tanto, el nodo no encaja con la expresión y se añade al árbol DOM.

3. Cuando aparece el segundo nodo *SPAN* que contiene un atributo *class* con valor *c2* (en este caso, se trata del último nodo SPAN), también se considera la sub-expresión *//TABLE[@id="table2"]/TR[2]*, que en este caso, sí que encaja con sus ascendientes consecutivos (el nodo *TABLE* y su segundo hijo *TR*).

   Una vez que se ha llegado a este punto, todos los fragmentos que componen la expresión XPath, han sido asignados a nodos del árbol DOM, con lo que este nodo *SPAN* se considerará un nodo irrelevante (raíz de un

subárbol descartable), y como consecuencia, este nodo y todos sus descendientes serán descartados y no se incluirán en el árbol DOM final.

3.2 EJECUCIÓN EN PARALELO DE CÓDIGO JAVASCRIPT

Los navegadores convencionales, ejecutan de manera secuencial la evaluación del código JavaScript contenido dentro de los documentos HTML.

Sobre este principio de funcionamiento, tan sólo existen dos excepciones, comentadas con anterioridad en el apartado 2.2.3:

1. Los scripts que contienen el atributo *async*.
2. Los Web Workers definidos en el estándar HTML5.

En este apartado, se detalla la técnica de optimización que permite la ejecución en paralelo de aquellos scripts de una misma página, que no tienen dependencias entre ellos.

Esta técnica, se basa en un análisis pormenorizado de la evaluación de scripts, durante la primera ejecución de la secuencia de navegación.

Durante esta primera ejecución, se va a monitorizar la ejecución de JavaScript para detectar aquellos nodos de tipo script, que no tienen dependencias entre ellos y de este modo, con posterioridad, podrían evaluarse de forma concurrente (durante las siguientes ejecuciones de la misma secuencia).

Por lo tanto, para el desarrollo de esta funcionalidad, el componente de navegación va a trabajar de manera muy similar a cómo lo hace para construir un árbol DOM minimizado (técnica explicada en el apartado anterior), con dos fases de funcionamiento bien diferenciadas:

1. Durante la fase de optimización, se calculará la información necesaria para poder identificar a los scripts, así como las dependencias que hay entre ellos.
2. Durante la fase de ejecución, se utilizará la información calculada con anterioridad, para ejecutar en paralelo los scripts detectados como independientes entre sí.

3.2.1 Detección de las dependencias entre los scripts

Como se describe en detalle en el apartado 3.1.2, durante la fase de optimización, la secuencia de navegación se ejecuta una sola vez, y se monitoriza la ejecución del código JavaScript, con el objetivo de obtener todas las dependencias entre todos los nodos del árbol DOM de la página web.

Las dependencias se calculan utilizando las reglas detalladas en el apartado 3.1.1 y tras la ejecución de la secuencia de navegación web, se genera un grafo que incluirá, para cada uno de los nodos del árbol DOM, todos aquellos otros nodos que son dependencias de éste.

Este mimo grafo, utilizado para la obtención de los nodos que representan fragmentos irrelevantes de la página HTML, también se va a utilizar para desarrollar la técnica de ejecución de scripts en paralelo.

Algorithm: builds the scripts dependency ghaph.
Inputs: 1. *dependencyGraph:* graph with the dependencies between DOM nodes.
2. *DOM:* DOM tree of the document.
Output: *scriptDependencyGraph* with the dependencies between script nodes.

```
1: function BuildScriptsDependencyGraph(dependencyGraph,  DOM) {
2:    var scriptDependencyGraph ← new Graph()
3:    for each node in DOM {
4:      if (node.isScript()) {
5:        AddScriptDependencies(node, dependencyGraph, node, scriptDependencyGraph);
6:        if(not scriptDependencyGraph.Contains(node)) {
7:          scriptDependencyGraph.AddSafeScript(node)
8:        }
9:      }
10:   }
11:   return scriptDependencyGraph
12: }
```

Algorithm: recursivelly collect *script* dependencies starting in the *node* dependencies.
Inputs: 1. *node*: node to check if contains *script* dependencies
2. *dependencyGraph*: graph with the dependencies between DOM nodes.
3. *script*: script node to add dependencies.
4. *scriptDependencyGraph*: i/o parameter with the dependencies between scripts.
Output: *script* dependencies will be added to *scriptDependencyGraph*.

```
1: function AddScriptDependencies(node, dependencyGraph, script, scriptDependencyGraph) {
2:   var nodeDependencies ← dependencyGraph.Get(node);
3:   for each nodeDependency in nodeDependencies {
4:     if (nodeDependency.isScript()) {
5:       scriptDependencyGraph.Add(script, nodeDependency)
6:     }
7:     AddScriptDependencies(nodeDependency,dependencyGraph,script,scriptDependencyGraph)
8:   }
9: }
```

Figura 51. Algoritmo para la construcción del grafo de dependencias entre scripts.

Para ello, tras la ejecución de la secuencia de navegación, y una vez generado el grafo con todas las dependencias entre los nodos, se procederá a generar una nueva estructura, más reducida, que sólo contendrá información acerca de las dependencias entre los nodos de tipo script, incluidos en el documento.

Esta estructura, se denominará grafo de dependencias entre scripts y contendrá, para cada uno de los scripts *S* presentes en la página HTML, una lista con otros scripts que deberán ejecutarse antes, para el correcto funcionamiento de *S*.

Si esos scripts no se ejecutasen antes que *S*, la evaluación del script *S* fallaría, con lo que produciría, con bastante probabilidad, el fallo en la ejecución de la secuencia de navegación web (podría generarse una página web incompleta o incorrecta).

El algoritmo de la Figura 51, describe el pseudo-código para la generación del grafo que contendrá todas las dependencias, entre los diferentes scripts presentes en la página HTML.

Las entradas de la función principal (*BuildScriptDependencyGraph*) serán, por un lado, el grafo con las dependencias entre todos los nodos del árbol DOM (*dependencyGraph*), calculado tras la ejecución de la secuencia de navegación web, durante la fase de optimización, y por otro lado, el árbol DOM de la página HTML.

El algoritmo itera por todos los nodos del árbol DOM de la página web, para localizar aquellos que son de tipo script.

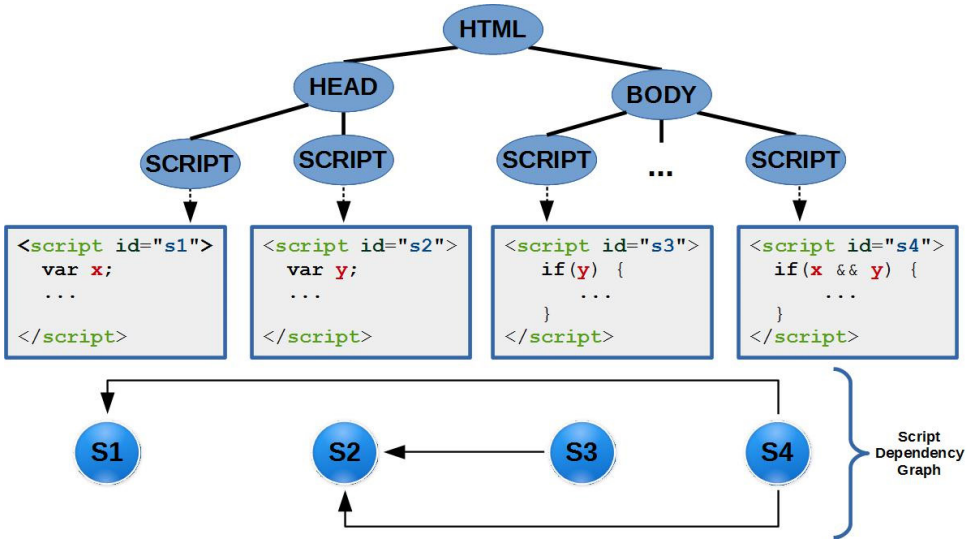


Figura 52. Ejemplo de grafo de dependencias entre scripts.

Cada vez que se identifica a uno de éstos, se realiza una búsqueda dentro del grafo de dependencias, para identificar otros scripts que sean dependencias de éste (utilizando la función *AddScriptDependencies*).

Por motivos de optimización que se explicarán más adelante, los scripts que no dependan de ningún otro script, se guardarán dentro de una lista especial (línea 7). Esta lista, se denominará lista de scripts seguros.

La Figura 52, muestra un ejemplo de generación de un grafo de dependencias entre scripts sobre un árbol DOM que contiene, entre otros nodos, cuatro de ellos de tipo script.

El primer script, identificado con el atributo *id* con valor *s1*, define una variable *x*. Este primer script, no tiene dependencias con los otros scripts presentes en la página.

El segundo script, identificado con el atributo *id* con valor *s2*, define una variable *y*. Este script tampoco tiene dependencias con otros scripts y por lo tanto, su ejecución se puede hacer en paralelo, con la del primer script *s1*.

El tercer script, identificado con el atributo *id* con valor *s3*, utiliza la variable *y*, definida con anterioridad en el script *s2*.

Por lo tanto, el grafo de dependencias entre scripts incluirá una dependencia entre *s3* y *s2*, lo que significa que *s3* no puede empezar su ejecución hasta que *s2* haya finalizado.

Por otro lado, *s3* no tiene ninguna dependencia con el primer script *s1*, con lo que la ejecución de éstos, sí que puede realizarse de manera concurrente.

Para finalizar, el cuarto script, identificado con el atributo *id* con valor *s4*, utiliza tanto la variable *x* como la variable *y*.

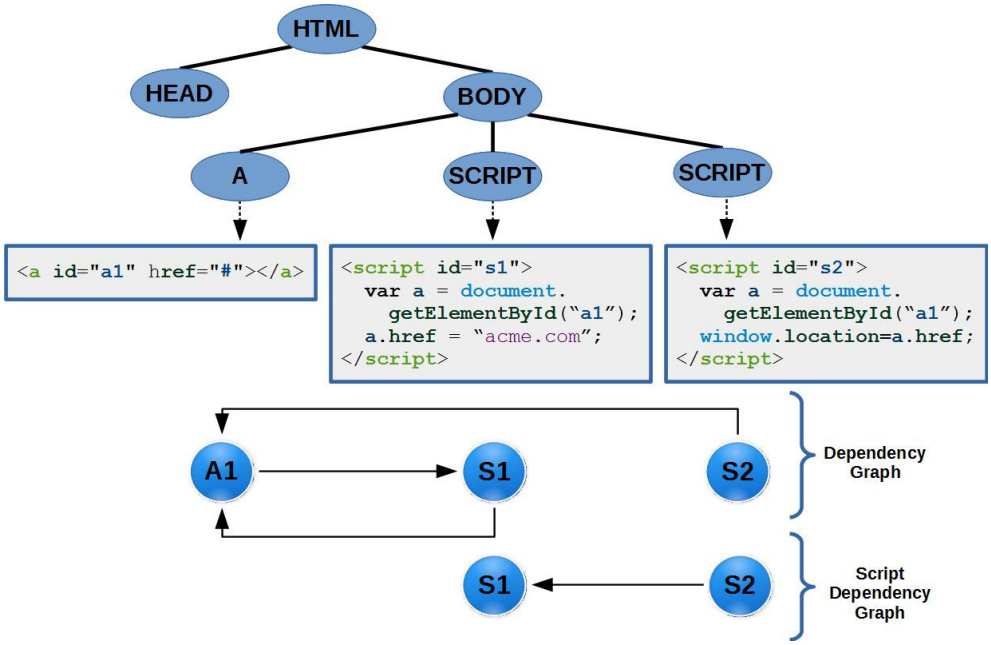


Figura 53. Ejemplo de grafo de dependencias entre scripts con nodos.

Por lo tanto, el grafo de dependencias entre scripts incluirá una dependencia de *s4* con *s1* y de *s4* con *s2*. Esto implica que *s4* no podrá paralelizarse ni con *s1* ni con *s2*, pero sí que podrá paralelizarse su ejecución con la de *s3*, al no existir una dependencia entre ellos.

La Figura 53, ilustra otro ejemplo de generación de grafo de dependencias entre scripts, pero en este caso, el proceso no sólo involucra a nodos de tipo script, sino también a otros nodos del árbol DOM de la página.

En este segundo ejemplo, se incluyen dos nodos de tipo script, identificados con el atributo id con valor *s1* y *s2* respectivamente y un nodo de tipo enlace, identificado con el atributo *id* con valor *a1*.

El primero de los scripts, identificado con el atributo *id* con valor *s1*, utiliza el nodo enlace *a1*, a través de la función predefinida *getElementById*, del objeto *document*.

Aplicando las reglas definidas con anterioridad para el cálculo de dependencias, *s1* → *a1*.

Pero además, el script *s1* también modifica el atributo *href* del enlace *a1*. Por lo tanto *a1* → *s1*.

El segundo de los scripts, identificado con el atributo *id* con valor *s2*, también utiliza el nodo enlace *a1*, con lo que *s2* → *a1*. Además, *s2* utiliza el atributo *href* del nodo *a1*. Este atributo, ha sido modificado con anterioridad durante la ejecución de *s1*.

Por lo tanto, en este caso, para la correcta ejecución del script *s2*, *s1* debe ser ejecutado antes, y no puede existir paralelización entre ellos. Esta situación se refleja en el grafo de dependencias entre scripts, el cual contendrá una dependencia *s2* → *s1*.

Una vez que se ha generado el grafo con las dependencias entre scripts, éste debe ser almacenado para su uso posterior, durante la fase de ejecución.

Para almacenar cada uno de los scripts presentes en el grafo de dependencias entre scripts, se utilizará el mismo proceso descrito en el apartado 3.1.3, y por lo tanto, se generará una expresión XPath para cada script.

3.2.2 Evaluación en paralelo durante la fase de ejecución

La fase de ejecución, involucra todas las ejecuciones posteriores de la misma secuencia de navegación, una vez que se ha calculado la información de optimización, durante la primera ejecución de la secuencia.

Recuérdese que, en el caso de la técnica de optimización para la ejecución en paralelo de código JavaScript, durante la fase de optimización, se realiza el cálculo del grafo de dependencias entre scripts.

Este grafo, se almacena utilizando expresiones XPath para representar a cada uno de los nodos de tipo script. Con posterioridad, este grafo será utilizado durante las siguientes ejecuciones de la misma secuencia para identificar elementos paralelizables.

El algoritmo de la Figura 54, detalla el pseudo-código de la función que chequea si un script presente en el árbol DOM de la página HTML, está preparado para su ejecución.

En primer lugar, el algoritmo comprueba si ese script es un script seguro (*IsSafeScript* en la línea 2), esto es, un script que no tiene dependencias con ningún otro script de la página HTML y que por lo tanto, durante la fase de optimización, ha sido almacenado en la lista especial de scripts sin dependencias.

En caso afirmativo, no se realiza ninguna comprobación adicional y el algoritmo determina que el script está preparado para su ejecución.

Si se trata de un script que tiene dependencias, se comprueba si todas estas dependencias han finalizado su ejecución (utilizando la función *HasFinished* en la línea 7).

Cuando todas las dependencias de un script han finalizado su ejecución, el algoritmo determina que éste está listo para su ejecución.

Algorithm: tests if a script is ready to run.
Inputs: 1. *script*: script node to test if it is ready to run.
2. *scriptDependencyGraph*: graph with dependencies between scripts.
Output: *true* if the script can be executed.

```
1:    function IsReadyForExecution(script, scriptDependencyGraph){
2:       if(scriptDependencyGraph.IsSafeScript(script)) {
3:           return true
4:       }
5:       var dependencies ← scriptDependencyGraph.Get(script);
6:       for each scriptDependency in dependencies {
7:           if (not scriptDependency.HasFinished()){
8:               return false
9:           }
10:      }
11:      return true
12:   }
```

Figura 54. Algoritmo para detectar si un script está preparado para ejecutarse.

Algorithm: executes one pending event in the input *thread*.
Inputs: 1. *thread*: idle thread ready to execute pending events.
2. *eventQueue*: queue containing pending events.
3. *scriptDependencyGraph*: graph with script dependencies.
Output: pending event is dispatched in the input *thread*.

```
1: function execute(thread, eventQueue, scriptDependencyGraph){
2:   if (eventQueue.IsEmpty()) return
3:   if (thread.IsMainThread()) {
4:     var event ← eventQueue.Pop()
5:     if (scriptDependencyGraph is null or not event.IsScriptEvent()) {
6:        thread.dispatch(event)
7:        return
8:     }
9:     var script ← event.GetScript()
10:    if (script.IsAsync() or
11:       IsReadyForExecution(script, scriptDependencyGraph) ){
12:       thread.dispatch(event)
13:    }
14:    return
15:  }
16:  for each event in eventQueue {
17:    if (not event.IsScriptEvent()) {
18:       continue
19:    }
20:    var script ← event.GetScript()
21:    if (script.IsAsync() or
22:       IsReadyForExecution(script, scriptDependencyGraph) ){
23:       thread.dispatch(event)
24:       return
25:    }
26:  }
27: }
```

Figura 55. Algoritmo para la ejecución de eventos.

El algoritmo de la Figura 55, detalla el proceso de selección de scripts para su ejecución (así como otros eventos de navegación, que se producen durante la ejecución de la secuencia de navegación).

Este algoritmo, está diseñado para un navegador a medida que contiene los siguientes componentes dentro de su motor de renderización:

1. Cola de eventos: almacena los eventos del navegador para su ejecución. Esta cola almacenará diferentes tipos de eventos, como por ejemplo:

- Procesado de código HTML.
- Evaluación de hojas de estilo CSS.
- Evaluación de nodos de tipo script.
- Ejecución de manejadores de acciones de usuario.
- Ejecución de funciones asíncronas, etc.

La ejecución de eventos, puede producir nuevos eventos que serán almacenados también en la cola de eventos.

2. *Thread* principal: ejecuta cualquier evento almacenado en la cola de eventos (ejecución de scripts, procesado de HTML, etc.).

   Si la secuencia de navegación se ejecuta sin hacer uso de la técnica de ejecución de scripts en paralelo, todos los eventos se procesarán de manera secuencial en este hilo, emulando el comportamiento síncrono de un navegador convencional.

3. Pool de *threads*: sólo ejecutan eventos de ejecución de scripts en paralelo.

   Este pool sólo se utiliza en caso de que la página HTML que se está cargando, tenga disponible el grafo con la información de las dependencias entre los scripts. En caso contrario, los scripts se ejecutarán de forma secuencial, en el *thread* principal.

El algoritmo de ejecución de eventos, recibe tres entradas:

1. El *thread* que va a ejecutar el evento.
2. La cola que almacena los eventos pendientes de ejecución.
3. Y el grafo con la información de las dependencias entre los scripts.

En primer lugar, el algoritmo comprueba si la cola de eventos pendientes contiene algún evento (*IsEmpty*, en la línea 2). El algoritmo se detiene en caso de que la cola esté vacía.

A continuación, el algoritmo comprueba si el *thread* llamante es el *thread* principal de ejecución.

En caso afirmativo, este *thread* ejecutará el primer evento en la cola de eventos (obtenido utilizando la función *Pop*, en la línea 4). El *thread* principal ejecuta eventos de cualquier tipo, incluidos eventos de evaluación de scripts.

En caso de que el primer evento almacenado en la cola sea un evento de evaluación de un script, se comprueba si este script está listo para ejecutarse.

Un script se considerará preparado para ejecutarse en dos escenarios:

1. El script contiene el atributo *async* (*IsAsync*, en la línea 10).

2. El script no tiene dependencias pendientes de finalización (*IsReadyForExecutión* en la línea 11, esta función ha sido detallada en el algoritmo de la Figura 54).

En caso de que el primer evento almacenado en la cola no sea un evento de evaluación de un script, entonces este evento se ejecuta de forma directa (*dispatch*, en la línea 12).

Si el *thread* llamante es un *thread* del pool, el algoritmo itera por los eventos almacenados en la cola, buscando eventos de ejecución de JavaScript (*IsScriptEvent*, en la línea 17).

El pool de *threads*, sólo ejecutará este tipo de eventos. Cuando aparece un evento de evaluación de JavaScript, se comprueba si está preparado para ejecutarse, y en caso afirmativo se procesa en este *thread* del pool (*dispatch*, en la línea 23).

Cuando un *thread* termina la ejecución de un evento, vuelve a invocar al algoritmo, para obtener y ejecutar un nuevo evento almacenado en la cola.

Cada vez que un script finaliza su ejecución (tanto los ejecutados en el *thread* principal como los ejecutados en el pool), el grafo de dependencias se actualiza marcando ese script como finalizado. Una vez hecho esto, otros scripts pueden modificar su estado y pasar a estar preparados para su ejecución.

Aunque los algoritmos para representar e identificar nodos en el árbol DOM de la página HTML han sido diseñados para resistir pequeños cambios en la página, en algunos escenarios, puede darse el caso de que un cambio en el código fuente, provoque que algunos nodos de tipo script no sean identificados de forma correcta, en la nueva versión del árbol DOM.

Cuando esto sucede, los scripts con dependencias que no han sido localizadas no se podrán paralelizar y serán ejecutados en el *thread* principal, tras finalizar los otros eventos almacenados en la cola, siguiendo un orden natural de ejecución (el orden de inserción en la cola)

Si en la página se detectan nuevos scripts, el grafo de dependencias se invalida y debe ser recalculado, recopilando de nuevo la información de optimización.

Los eventos de evaluación de JavaScript que se ejecutan en los *threads* auxiliares del pool, se pueden paralelizar con otro tipo de eventos que se ejecutan en el *thread* principal del motor de renderización, como por ejemplo, el procesado incremental del flujo de datos HTML.

Cuando esto se produce, es necesario tener en cuenta algunas situaciones especiales, durante la fase de ejecución, como la que se describe a continuación en la Figura 56.

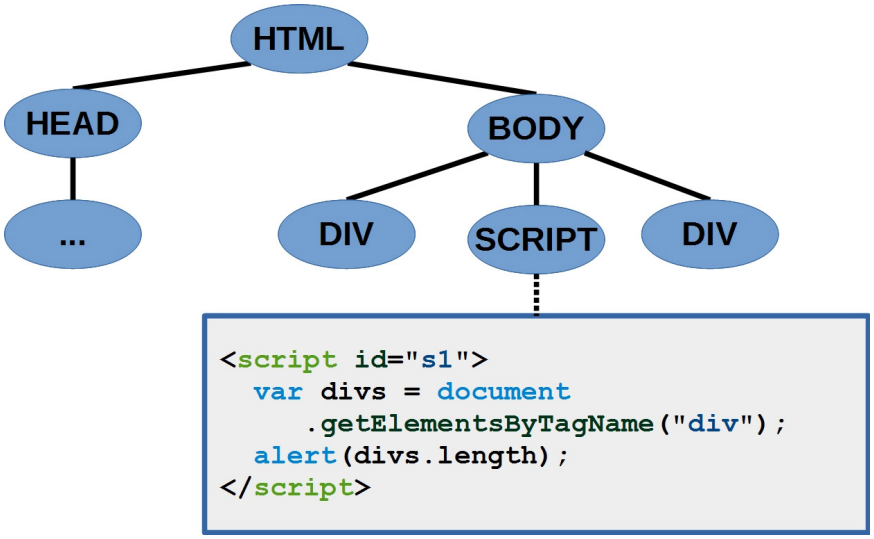


Figura 56. Ejemplo de paralelización del procesado HTML con la evaluación de scripts.

El árbol DOM del ejemplo, contiene dos nodos de tipo *DIV*, uno de ellos se encuentra situado en el árbol antes que el script *s1* y otro, se encuentra situado en el árbol después del script.

Durante el proceso de carga del árbol DOM de la figura, el nodo script *s1* se podría ejecutar en uno de los *threads* auxiliares del pool y la evaluación de este script, se ejecutaría en paralelo con el procesamiento del árbol DOM.

Si el proceso de carga del árbol DOM se ejecutase de manera secuencial (sin utilizar la técnica de evaluación de JavaScript en paralelo), la evaluación del script *s1*, sólo podría acceder al primero de los nodos de tipo *DIV*, dado que el segundo de ellos se encuentra en una posición posterior en el árbol. Esto significa que cuando el script se está ejecutando, todavía no se ha añadido el segundo nodo DIV al árbol DOM.

Cuando el proceso de carga de la página utiliza la técnica de evaluación de JavaScript en paralelo, el comportamiento tiene que ser el mismo que cuando la carga de la página es secuencial, aunque en este caso, al ejecutarse en paralelo el procesamiento del flujo de datos HTML con la evaluación del script, el segundo nodo de tipo *DIV* puede haber sido añadido al árbol DOM cuando todavía el script se está ejecutando.

Para prevenir problemas derivados de esta situación, el motor de renderización de un navegador que implemente esta técnica, debe utilizar un mecanismo de bloqueo de nodos durante la fase de ejecución, por ejemplo, en base a identificadores numéricos. Cada vez que se añade un nuevo nodo al árbol DOM de la página, a este nodo se le asigna un identificador numérico. Estos identificadores se asignan de manera consecutiva y durante la evaluación de un script, el contexto de ejecución de este script, se limitará a los nodos con un identificador menor que el identificador del propio script.

De este modo, en el ejemplo de la Figura 56, cuando el script utiliza la función *getElementsByTagName*, para obtener todos los nodos con etiqueta *DIV*, esta función devolverá sólo uno de los nodos *DIV*, el que se encuentra situado en el árbol en una posición anterior (su identificador numérico será menor que el identificador numérico del propio nodo script).

La Figura 57, muestra una simulación mediante un cronograma, de los eventos generados en un navegador a medida cuando utiliza la técnica de ejecución de scripts en paralelo. Esta figura, muestra el proceso de carga de la misma página web que se mostraba en la Figura 11, en el cronograma de eventos de Google Chrome.

Los scripts contenidos en la página, han sido identificados como scripts seguros (sin dependencias entre ellos) y por lo tanto, su ejecución se podrá realizar en paralelo.

Las llamadas de red se representan en las acciones *"Send Request"*, los eventos de procesado de HTML se representan en las acciones "*Parse HTML"*, y los eventos de evaluación de JavaScript, se representan en las acciones "*Evaluate Script"* y "*Function Call"*.

El proceso de carga, involucra la descarga de los siguientes elementos:

1. Hojas de estilo CSS.
2. Tres ficheros JavaScript referenciados desde el código fuente HTML de la página.
3. Y además, un cuarto fichero JavaScript, generado de manera dinámica, durante la evaluación del script *github.js*.

Durante el procesado del código fuente HTML, el analizador descubre los objetos externos de manera incremental, e inicia de forma automática la descarga de estos ficheros (estas descargas se realizarán en paralelo).

Además, los scripts correspondientes a los tres ficheros JavaScript, también se evaluarán en paralelo, dado que durante la fase de optimización, se ha detectado que no existe ninguna dependencia entre ellos.

Incluso el orden en el que se ejecutan, puede ser diferente del orden natural de ejecución que se muestra en la Figura 11. Cada uno de estos scripts, se puede ejecutar en un *thread* diferente.

En la Figura 57, los eventos de ejecución de scripts resaltados con líneas de trazos discontinuos, representan tiempos de ejecución en los que se han ejecutado varios scripts en paralelo.

Los eventos "*Function Call x 4"* y "*Function Call x 9"*, se han generado de forma dinámica, durante la evaluación del script *frameworks.js*. Para determinar si estos eventos se pueden paralelizar, se utilizarán las dependencias asociadas al nodo script desde el que se han creado estos eventos (en este caso, el nodo script asociado al fichero *frameworks.js*).

"Function Call x 4" se ejecuta en paralelo con el script *github.js*.

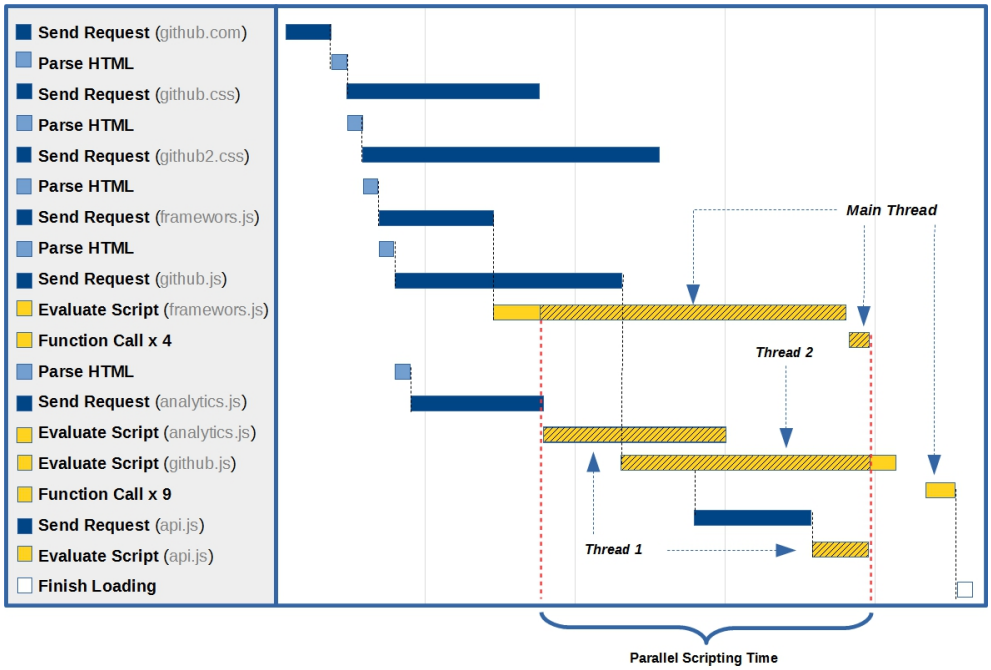


Figura 57. Ejecución de eventos en el navegador a medida.

"*Function Call x 9"*, comienza su ejecución tras un intervalo de tiempo, dado que este evento ha sido generado con una función temporizador (en este caso, ha sido iniciado con la función predefinida *setTimeout*).

El script *api.js* ha sido generado de forma dinámica (utilizando la función predefinida *write* del objeto *document*) y además, este script también se puede paralelizar al contener el atributo *async* (como no tiene dependencias con los otros scripts, se podría paralelizar igualmente aunque no tuviese el atributo async).

3.3 EVALUACIÓN DE ESTILOS CSS BAJO DEMANDA

En este apartado, se detalla la tercera de las optimizaciones propuestas en este trabajo. En este caso, se trata de la técnica de evaluación de estilos CSS bajo demanda.

Como se ha comentado en los apartados anteriores, los navegadores a medida desarrollados de forma específica para tareas de automatización web, sólo acostumbran a trabajar con el árbol DOM del documento HTML.

El diseño de la página, que contiene información con los atributos visuales necesarios para dibujar cada elemento en la pantalla, no es necesario en los navegadores ad-hoc, dado que éstos, no están desarrollados para su utilización por parte de personas y por lo tanto, no requieren de la renderización de la página HTML en la ventana de visualización.

Sin embargo, los atributos de estilo, pueden ser referenciados durante la ejecución del código JavaScript. Esto significa que el navegador a medida debe desarrollar un subsistema de emulación de CSS, que sea capaz de calcular esta información de visualización. En la siguiente figura, se ilustra un ejemplo de este escenario.

En este ejemplo, durante la ejecución de JavaScript se accede al atributo CSS *display*, que indica si un nodo del árbol DOM es visible, o por el contrario, está oculto.

Si el navegador a medida no fuese capaz de calcular el valor correcto de este atributo, la condición de la línea 13 de la figura, podría no ser evaluada de forma correcta, con lo que el resultado de la ejecución de ese script, sería diferente en el navegador a medida y en el navegador convencional.

Para emular el soporte de CSS de los navegadores convencionales, con la técnica diseñada en este trabajo, la información de visualización estará almacenada en cada uno de los nodos del árbol DOM construido por el navegador.

Cada uno de estos nodos, contendrá un objeto con los atributos de visualización que pueden ser utilizados desde el contexto de ejecución de JavaScript.

Este objeto, en el que se almacena la información de visualización, inicialmente, será nulo para todos los objetos del árbol DOM y durante la ejecución de JavaScript, en el momento en el que se accede a estos datos, el objeto se crea y se calculan de forma dinámica, los valores de los atributos necesarios.

De esta forma, la aplicación de las reglas contenidas en las hojas de estilo CSS, se ejecuta sólo para aquellos nodos en los que es necesario, con la consiguiente mejora, tanto el uso de CPU, como en el uso de memoria.

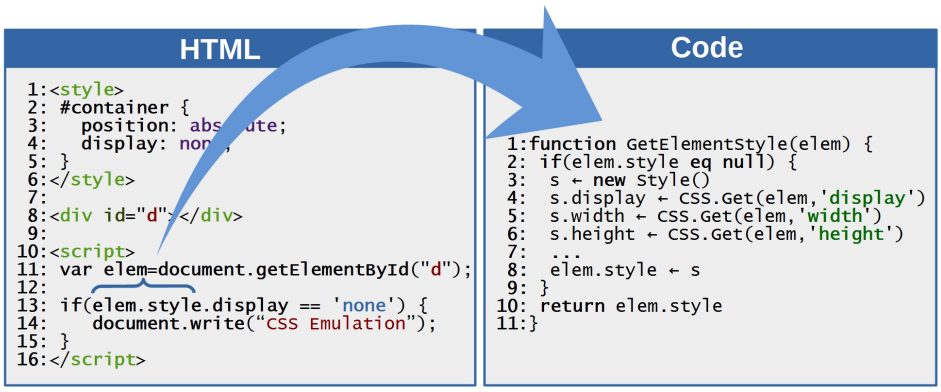


Figura 58. Subsistema de simulación de CSS.

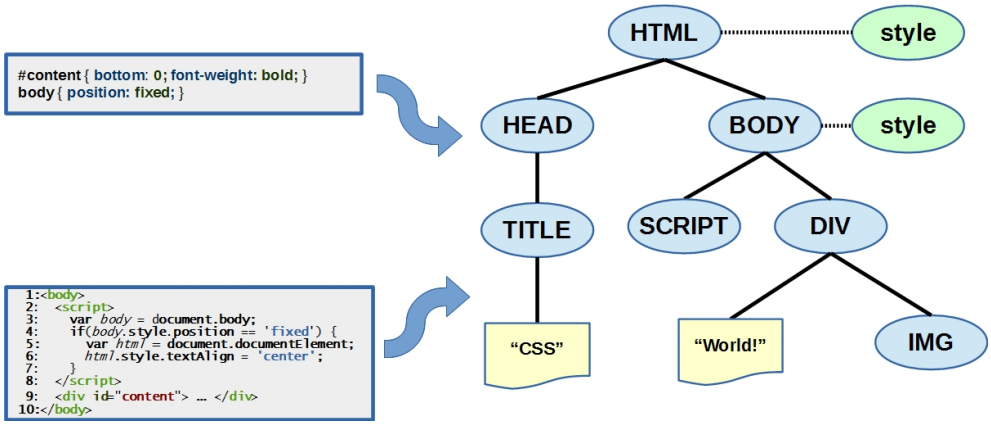


Figura 59. Procedimiento de creación de objetos de visualización.

Para los objetos en los que no es necesario calcular la información de CSS, el objeto con la información de visualización permanecerá con valor nulo.

La Figura 59, ilustra un pequeño ejemplo en el que, de todos los nodos que forman parte del árbol DOM, tan sólo el nodo *HTML* y el nodo *BODY* requieren de la generación de los objetos de visualización.

A continuación, se muestran algunos de los objetos predefinidos y funciones JavaScript que dispararán llamadas al subsistema de simulación de CSS:

1. Objeto *style* de los nodos del DOM.
2. Objeto *computedStyle* de los nodos del DOM.
3. Función *getElementsByClassName* del objeto *document*.
4. Otros atributos de lectura/escritura, predefinidos en los elementos del árbol DOM, por ejemplo *offsetWidth*, *clientWidth*, *scrollTop*, etc.

Esta optimización para dar soporte a la evaluación de estilos CSS bajo demanda, afectará a la arquitectura final de un componente de navegación que la utilice, como se observará en el siguiente capítulo.

En esta arquitectura, el subsistema de evaluación de CSS, sólo se utilizará desde el motor de evaluación de JavaScript. Además, el componente de navegación debe ser capaz de descargar y evaluar hojas de estilo.

4 DESCRIPCIÓN DE LA ARQUITECTURA DE UN COMPONENTE DE NAVEGACIÓN OPTIMIZADO PARA LA EJECUCIÓN EFICIENTE DE SECUENCIAS DE NAVEGACIÓN WEB

En este capítulo, se describe la arquitectura diseñada para un componente de navegación a medida con las siguientes características:

1. Carece de interfaz de usuario.
2. Está especializado en la ejecución de tareas de automatización web.
3. Proporciona soporte para las distintas técnicas de optimización detalladas en el capítulo anterior:
    - Generación de un árbol DOM minimizado.
    - Ejecución en paralelo de scripts sin dependencias entre ellos.
    - Evaluación de estilos CSS bajo demanda.

En los siguientes apartados, en primer lugar, se mostrará un resumen de los elementos más importantes que forman parte de la arquitectura del componente de navegación y posteriormente, se detallará en profundidad cada uno de ellos.

A continuación, se describe de forma breve, en qué consiste el mecanismo de configuración manual del componente de navegación.

Y por último, se muestran las etapas de procesamiento por las que pasa el motor de renderización.

4.1 COMPONENTES QUE FORMAN LA ARQUITECTURA DEL NAVEGADOR

En este apartado, se muestra una visión general de los distintos componentes que integran la arquitectura del sistema de navegación.

La arquitectura propuesta, se ve afectada por las optimizaciones descritas con anterioridad, de la siguiente manera:

1. Para dar soporte a la técnica de construcción de un árbol DOM minimizado, es necesario proporcionar un subsistema capaz de generar y almacenar la información de optimización con los fragmentos irrelevantes identificados. Además, en tiempo de ejecución, este subsistema se utilizará para construir una versión minimizada del árbol, siempre y cuando haya información de optimización disponible.

2. Para dar soporte a la técnica de ejecución de JavaScript en paralelo, también es necesario un subsistema capaz de generar la información de optimización con el grafo de dependencias entre scripts y además, es necesario que la arquitectura proporcione un mecanismo *multi-thread*, que permita evaluar varios scripts al mismo tiempo durante la fase de ejecución.

3. Para dar soporte a la técnica de evaluación de estilos CSS bajo demanda, se accederá al subsistema CSS tan solo durante la evaluación del código JavaScript y además, la información con los atributos de visualización, se almacenará dentro de los propios nodos del árbol DOM.

En la Figura 60, se muestran los elementos de alto nivel que forman parte de la arquitectura del sistema de navegación.

Como se puede observar, existen similitudes con la arquitectura de referencia de los navegadores convencionales, pero se añaden componentes específicos, que permiten dar soporte a las diferentes técnicas de optimización diseñadas en este trabajo.

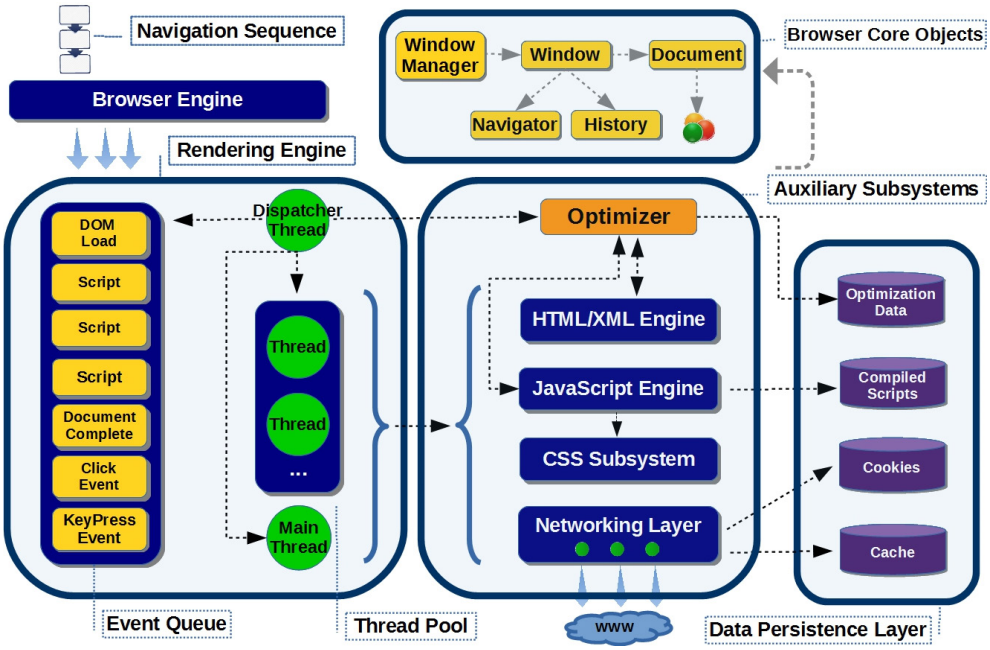


Figura 60. Arquitectura del componente de navegación.

Los componentes de más alto nivel son los siguientes:

1. Motor de navegación (*Browser Engine*).
2. Motor de renderización (*Rendering Engine*).
3. Subsistemas auxiliares (A*uxiliary Subsystems*).
4. Capa de persistencia de datos (*Data Persistence Layer*).
5. Objetos del navegador (*Browser Objects*).

Dentro de este sistema de navegación, el motor de renderización es el elemento más importante, y representa el núcleo central a la hora de procesar los contenidos HTML descargados de la red.

Los subsistemas que forman parte de este motor de renderización son los siguientes:

1. *Thread* principal para la ejecución de cualquier tipo de evento (*Main Thread*).
2. Pool de *threads* para la ejecución de JavaScript en paralelo (*Thread Pool*).
3. Cola de eventos pendientes de ejecución (*Event Queue*).
4. Despachador de eventos (*Dispatcher*).

Además, el motor de renderización hace uso de los distintos subsistemas auxiliares:

1. Subsistema de procesado de XML y HTML (*HTML/XML Engine*).
2. Optimizador (Optimizer).

3. Motor de ejecución de JavaScript (*JavaScript Engine*).
4. Subsistema de emulación de CSS (*CSS Engine*).
5. Capa de acceso a la red (*Networking Layer*).

Tanto el motor de renderización, como los subsistemas auxiliares, pueden acceder a los objetos en memoria del navegador. Entre estos objetos, destacan sobre todo el árbol DOM de los diferentes documentos que han sido descargados y procesados, así como otros objetos predefinidos que se detallarán más adelante.

4.2 MODELO DE *THREADS* DE LA ARQUITECTURA DEL COMPONENTE DE NAVEGACIÓN

En un entorno de funcionamiento básico, sin ejecución de JavaScript en paralelo, el componente de navegación utiliza tan solo dos *threads* (en este caso, se trata de hilos ligeros y no de procesos del sistema operativo).

En el primero de ellos, reside el motor de navegación (*Browser Engine Thread*, en la Figura 61) y en el segundo, reside el *thread* principal con el motor de renderización (*Main Thread* en la Figura 61). Este último, es el encargado de procesar los distintos eventos, y de realizar la carga de los documentos HTML.

En este modelo de funcionamiento básico, todos los eventos se procesan de forma secuencial, de uno en uno, siguiendo el orden natural de inserción (excepto algunos casos particulares que se detallan más adelante). En este caso, todos los eventos se ejecutan en el *thread* principal de ejecución.

De manera adicional, también se utilizará un pool de *threads* auxiliares en la capa de red, para permitir múltiples descargas en paralelo.

El nexo de unión entre los dos *threads* principales, es la cola de eventos pendientes de ejecución.

En esta cola, el motor de navegación insertará los eventos generados a partir de la traducción de la secuencia de navegación web, y el *thread* principal leerá esos eventos de la cola, para procesarlos y ejecutarlos de uno en uno, siguiendo el orden natural de inserción en la cola.

En apartados posteriores, se describen algunos de los tipos de eventos más relevantes que puede ejecutar el componente de navegación.

Cabe destacar, que en este entorno de funcionamiento básico, el *thread* principal accede de forma directa a la cola de eventos.

Como se detallará más adelante, en un entorno de ejecución optimizado, para evaluar los scripts en paralelo, la asignación de eventos a *threads* se centralizará en un componente dedicado y por lo tanto, el *thread* principal no accederá de manera directa a la cola de eventos.

Como consecuencia de la ejecución de un evento, se pueden generar nuevos eventos y es el propio *thread* principal, el encargado de añadirlos a la cola.

Por ejemplo, durante la ejecución de un nodo de tipo script, se puede generar de forma dinámica un evento "*click*", sobre alguno de los nodos del árbol DOM.

La Figura 61, ilustra este modelo de funcionamiento básico, cuando el componente de navegación opera de manera secuencial, sin utilizar las capacidades de evaluación de JavaScript en paralelo.

En la figura, se pueden observar los dos *threads* principales y su mecanismo de intercomunicación a través de la cola de eventos.

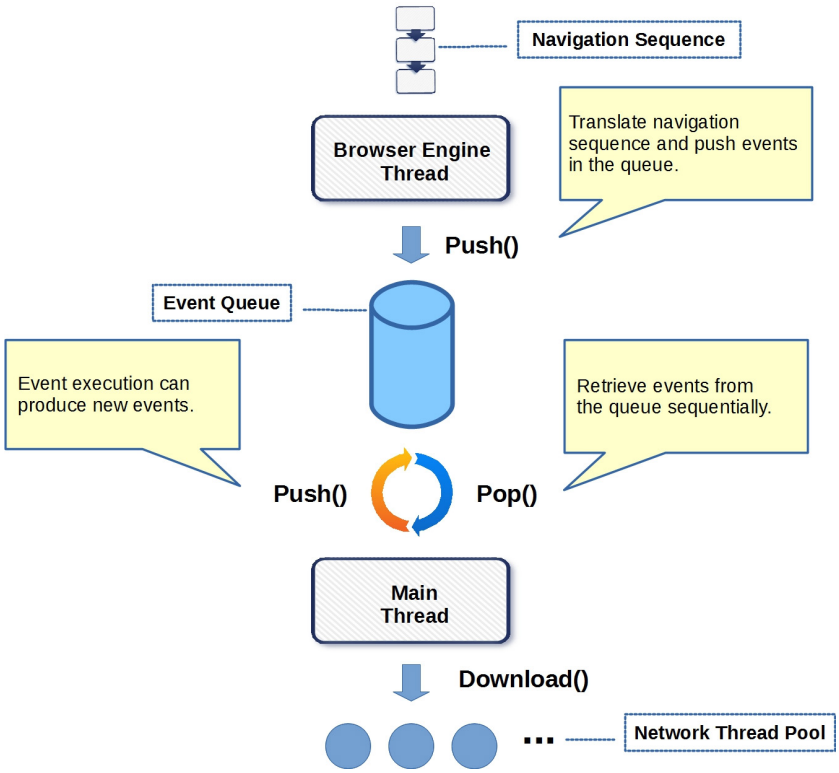


Figura 61. Procesamiento básico en el componente de navegación.

En este escenario, el *thread* principal accede a la cola de eventos pendientes y siempre obtiene el primer evento almacenado en la cola (utiliza el método de acceso "*Pop*"). Por lo tanto, los eventos se obtienen y ejecutan de manera secuencial, siguiendo el orden natural en el que han sido insertados.

Más adelante, se podrá observar como este principio de funcionamiento general tiene algunas excepciones, en función de los distintos estados por los que pueden pasar los eventos almacenados en la cola.

Cuando el componente de navegación opera en un entorno de ejecución optimizado, con información para ejecutar scripts en paralelo, el número de *threads* que intervienen durante el procesamiento de la página web, puede verse incrementado de manera considerable.

Para ello, la arquitectura del componente de navegación proporciona un pool de *threads* reutilizables (*Script Execution Thread Pool* en la Figura 62). En el diseño actual, en este pool sólo se ejecutan eventos de evaluación de código JavaScript.

Adicionalmente, en este segundo escenario, también interviene un nuevo elemento de la arquitectura: el *thread* despachador de eventos (*Dispatcher Thread*, en la Figura 62).

Este componente, es el encargado de asignar eventos pendientes de ejecución, tanto al *thread* principal, como a los *threads* del pool, y para ello, analiza la cola de eventos, buscando scripts que estén preparados (véase el algoritmo de la Figura 55).

Los scripts preparados, se pueden ejecutar en alguno de los *threads* disponibles en el pool dedicado a la ejecución de scripts (y también en el *thread* principal, cuando el primer evento pendiente en la cola, es de este tipo).

En este segundo entorno, todos los *threads* que intervienen en la ejecución de eventos (incluido el *thread* principal), obtienen el siguiente evento que van a ejecutar, a través del despachador de eventos. De esta forma, se garantiza un acceso a la cola de eventos centralizado y controlado, dentro de todo el motor de renderización.

Para buscar eventos de evaluación de JavaScript preparados para ejecutarse (cuando todas sus dependencias han finalizado), el *thread* despachador puede acceder al siguiente evento en la cola (por orden natural de inserción, método "*Pop*"), o puede acceder a cualquier evento pendiente de ejecución, independientemente de la posición que ocupe en la cola (iterando por todos ellos, método "*Get*").

El algoritmo utilizado para asignar eventos pendientes a *threads*, ha sido detallado en la Figura 55.

Los *threads* del pool dedicado a la ejecución de JavaScript en paralelo, también podrán añadir nuevos eventos a la cola, cuando estos eventos se han generado durante la evaluación de JavaScript.

La Figura 62, ilustra las interacciones entre los distintos *threads* del sistema, cuando el componente de navegación opera en un entorno multihilo, con soporte para evaluación de JavaScript en paralelo.

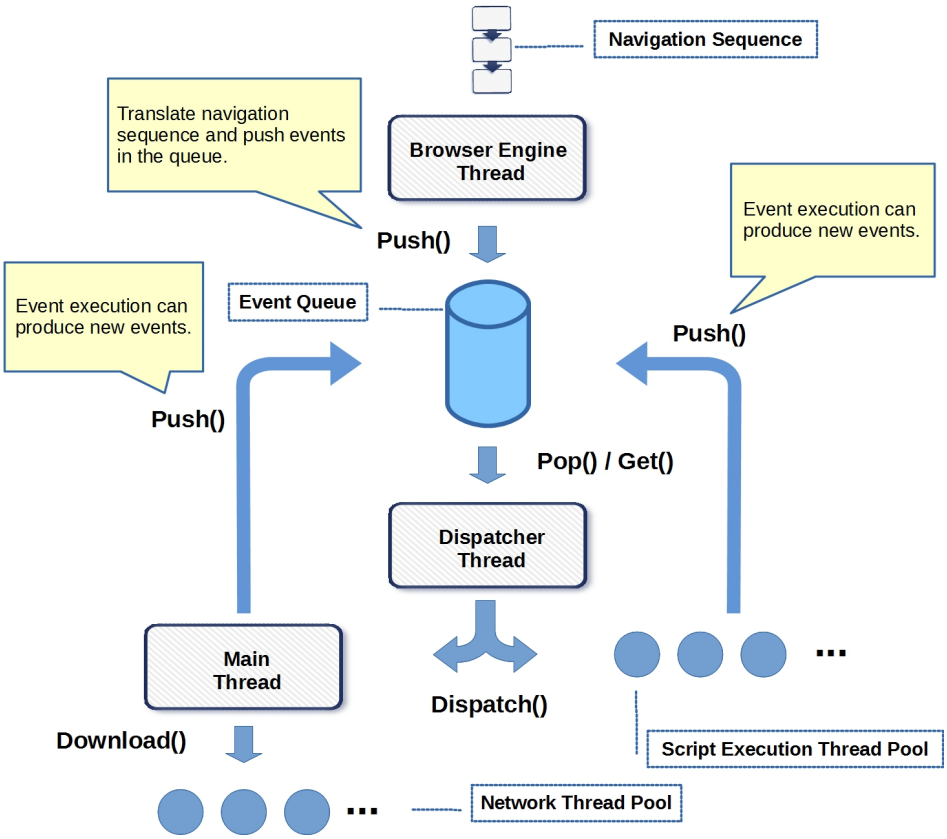


Figura 62. Procesamiento avanzado para dar soporte a la ejecución de JavaScript en paralelo.

En la figura, se puede observar cómo es el despachador de eventos, el que obtiene el evento a ejecutar y se lo asigna al *thread* correspondiente.

Para el procesamiento de los diferentes tipos de eventos, tanto el *thread* principal, como el pool de *threads* dedicado a la ejecución de scripts, hacen uso de los otros subsistemas auxiliares (subsistema de procesado de HTML y XML, subsistema de evaluación de CSS, motor de JavaScript, subsistema de red y optimizador).

En los siguientes apartados, se describen en profundidad cada uno los subsistemas que forman parte de la arquitectura, y que intervienen en algún punto del proceso de construcción del documento HTML.

4.3 MOTOR DE NAVEGACIÓN (BROWSER ENGINE)

El motor de navegación, es el componente encargado de recibir y gestionar los comandos del exterior, para traducirlos a elementos de procesamiento interno.

En un navegador convencional, el motor de navegación ofrece un API de alto nivel para interactuar con el motor de renderización. Este API permite operaciones tales como: navegar a una URL, recargar la página, navegar a la página anterior, etc.

En el caso de un navegador a medida, desarrollado para ejecutar tareas de automatización web, la entrada del motor de navegación es la secuencia de navegación web que se va a automatizar.

La tarea principal del motor de navegación, es la de realizar un procesado de esta secuencia de navegación, para traducir cada uno de los comandos que la forman, a eventos de navegación ejecutables dentro del motor de renderización.

Para interactuar con el motor de renderización, la arquitectura propuesta proporciona una cola, en la que se almacenarán los eventos pendientes de ejecución.

Cada comando que forma parte de la secuencia de navegación web, será traducido en el motor de navegación, a uno o varios eventos, que se almacenarán en la cola de eventos pendientes, para su posterior ejecución.

Además de traducir los comandos de la secuencia a eventos del motor de renderización, el motor de navegación ofrece otras funcionalidades relacionadas con la ejecución de la secuencia.

Entre estas funcionalidades destacan las siguientes:

1. Informar del estado actual de la ejecución de la secuencia (en curso, terminada, etc.), a las capas superiores.

   Para ello se aplican los siguientes principios de funcionamiento:

   a) El motor de navegación inserta los eventos en la cola, de uno en uno, y mantendrá una referencia a cada uno de los eventos insertados.

   b) En estos mismos eventos, el motor de renderización guardará información relevante sobre el proceso de ejecución.

   c) Antes de que el motor de navegación inserte un nuevo evento, el anterior debe finalizar su ejecución.

   d) El motor de renderización, marca los eventos como finalizados una vez que han sido completados.

   e) Los eventos están jerarquizados, es decir, si durante la ejecución de un evento se generan otros nuevos, el evento raíz no se marcará como completamente finalizado (estado completado), hasta que

terminen todos sus hijos (los distintos estados por los que puede pasar un evento, se describen más adelante).

2. Detener la ejecución pasado un cierto tiempo. Para ello, se insertará un evento especial en la cola de eventos, que le indica al motor de renderización que debe detener la ejecución y descartar todos los demás eventos pendientes.

3. Informar del motivo de error, en el caso de que la ejecución de alguno de los comandos haya fallado (por ejemplo, cuando se ejecuta una navegación a una página que no existe). Esta información se almacena en el propio evento, para que pueda ser accedida de manera sencilla desde el motor de navegación.

4.4 MOTOR DE RENDERIZACIÓN: EVENTOS Y COLA DE EVENTOS PENDIENTES

La cola de eventos, contiene una lista ordenada de eventos que están pendientes de ejecución.

Al principio, esta cola está vacía y los primeros eventos se introducen durante el proceso de traducción de la secuencia de navegación web, realizado en el motor de navegación.

Existen diferentes tipos de eventos almacenables en la cola y la ejecución de algunos de ellos, puede generar eventos adicionales, que también serán almacenados en esta misma cola. Cuando se produce esto, el evento raíz mantiene una referencia a todos sus eventos hijos. Por lo tanto, existe una estructura jerárquica en forma de árbol de eventos.

Un evento no se considerará finalizado, mientras no finalicen todos sus eventos hijos.

Cada evento, almacenará la siguiente información:

- Estado actual y transiciones previas entre estados.
- Evento padre que lo ha creado (en caso de que exista).
- El instante de creación y de inserción en la cola.
- La causa del error (si procede).
- Otro tipo de información, asociada a cada tipo específico de evento.

  Por ejemplo, un evento de navegación, almacena la siguiente información:

  - Ventana en la que se ha cargado el documento.
  - URL de la petición.
  - Tipo de petición (GET o POST).
  - Y en caso de petición POST, almacenará también el cuerpo del mensaje.
  - Cookies enviadas en la petición HTTP y cookies enviadas por el servidor web en la respuesta.

4.4.1 Estados de los eventos

Los diferentes estados por los que pueden pasar los eventos, son los siguientes:

1. No preparado (bloqueante): el evento se ha añadido a la cola, pero todavía no se puede ejecutar, por estar pendientes de completarse ciertas acciones asociadas.

   Por ejemplo, un evento de evaluación y carga de un documento CSS, se mantendrá en este estado mientras no finalice la descarga del documento

asociado (las descargas se ejecutan en paralelo, utilizando un pool de *threads* específicos en la capa de red).

Estos eventos, bloquean la cola y no se permite la ejecución de otros eventos (excepto scripts que se puedan evaluar en paralelo).

También es importante destacar que este estado no mantiene relación con la técnica de evaluación de JavaScript en paralelo. Cuando se utiliza esta técnica, lo que se revisa es si han finalizado los scripts que son dependencias de un script *S* (pero este script *S* se encuentra en estado preparado, porque ha finalizado la descarga asociada).

2. No preparado (no bloqueante): similar al estado anterior, salvo que este evento no bloquea la ejecución y permite ejecutar los eventos almacenados en posiciones posteriores, dentro de la cola de ejecución.

   Un ejemplo de esta situación, son los eventos de ejecución de JavaScript disparados con temporizadores. Estos eventos se ejecutan pasado un intervalo de tiempo preestablecido, y mientras tanto, el componente de navegación puede realizar otras acciones.

3. Preparado: el evento se ha añadido a la cola y está listo para ejecutarse.

   Si se trata de un evento de ejecución de JavaScript en paralelo, este evento se podrá ejecutar en un *thread* del pool, cuando todos los otros scripts que son dependencias de éste, han finalizado su ejecución y se encuentran en estado completado (siguiendo el gráfico de dependencias entre scripts).

   Si se trata de cualquier otro tipo de evento, entonces se ejecutará en el *thread* principal.

4. En ejecución: el evento sale de la cola y está siendo ejecutado. Si durante la ejecución de este evento se generan otros nuevos, este evento raíz mantendrá una referencia a todos sus eventos hijos.

   La mayoría de los eventos, no vuelven a la cola una vez finalizada su ejecución, excepto algunos casos especiales:

   - Los eventos de ejecución de JavaScript disparados con temporizadores periódicos (por medio de la función predefinida, *setInterval*), vuelven a la cola en estado no preparado (no bloqueante), hasta que vuelva a transcurrir, de nuevo, el tiempo de espera del temporizador.

   - Algunos eventos, generan otros nuevos, que deben procesarse de inmediato, incluso antes de finalizar la ejecución del evento en curso.

     En este caso, los eventos hijos generados, se insertan en la cola y el evento en ejecución se detiene y vuelve de nuevo a la cola,

en una posición posterior a la de los hijos que se han creado. Este evento en curso, continuará su ejecución cuando sus eventos hijos hayan finalizado.

Un ejemplo representativo de esta situación, es cuando aparece una hoja de estilo CSS dentro del flujo de datos (por ejemplo, durante el procesamiento del código fuente HTML).

En este caso, el procesamiento de la hoja de estilos CSS debe realizarse de inmediato, antes de continuar procesando el resto de datos HTML, por lo que:

- Se crea el evento de procesamiento de la hoja de estilos CSS y se inserta en la cola.
- El evento de procesamiento de datos HTML (evento en curso) se detiene y se vuelve a insertar en la cola, justo en la posición posterior a la del nuevo evento de procesado de CSS.

5. Finalizado: el evento ha terminado de manera correcta, pero durante su ejecución se han creado nuevos eventos, y éstos todavía están pendientes de finalización.
6. Completado: el evento y todos sus hijos han finalizado de forma correcta.
7. Fallido: el evento o alguno de sus eventos hijos no han finalizado con éxito.

4.4.2 Transiciones entre los estados de los eventos

La Figura 63, detalla las distintas transiciones entre los diferentes estados que un evento puede alcanzar dentro del componente de navegación.

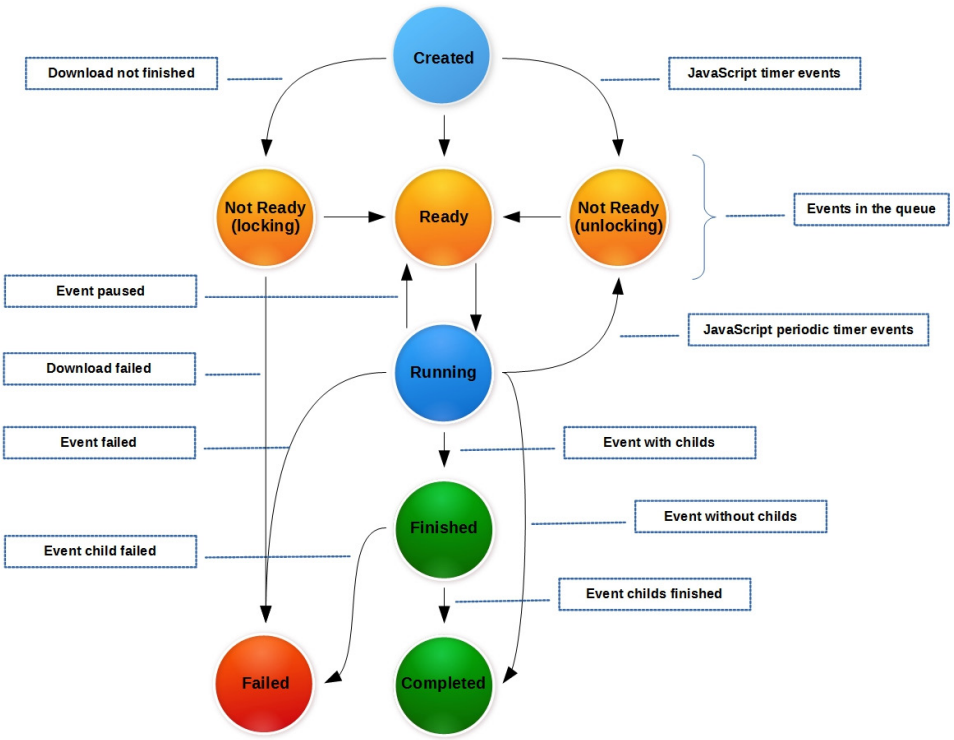


Figura 63. Transiciones de estados entre eventos.

Una vez que se crea el evento, este se almacena en la cola. Su estado inicial será *preparado*, si el evento está listo para ejecutarse o estado *no-preparado* (*bloqueante* o *no-bloqueante*), cuando todavía no se puede ejecutar.

En el caso de los eventos asociados a alguna descarga, el estado en el que se guardarán, será en estado *no-preparado* (*bloqueante*), a la espera de que finalice la descarga asociada. La transición que se realiza una vez que la descarga ha finalizado, es al estado *preparado*, siempre y cuando la descarga haya finalizado de manera correcta. En caso de que la descarga haya finalizado con error, el estado del evento pasará a *fallido*.

En el caso de eventos JavaScript asociados a un temporizador, su estado inicial será *no-preparado* (*no bloqueante*), mientras no se consuma el tiempo de espera asignado al temporizador. Una vez finalizado este tiempo de espera, el estado de estos eventos pasa a ser *preparado*.

Cuando el primer evento en la cola es de tipo *no-preparado bloqueante*, la ejecución se detiene a la espera de que este evento pase a estado *preparado* (o *fallido*).

Por ejemplo, si se trata de un evento de ejecución de JavaScript (en un entorno de funcionamiento básico, sin evaluación en paralelo), los diferentes scripts tienen que ejecutarse, de forma estricta, por orden de aparición en el documento HTML.

Una vez que los eventos pasan al estado *preparado* (siempre dentro de la cola), es cuando se pueden ejecutar.

En un entorno de funcionamiento básico, sin información para evaluar scripts en paralelo, los eventos se ejecutan de uno en uno y en un entorno de ejecución de JavaScript en paralelo, varios eventos de evaluación de JavaScript se pueden ejecutar al mismo tiempo.

Cuando un evento sale de la cola para ejecutarse, su estado pasa a ser *en ejecución*.

Si durante la ejecución de un evento, se generan nuevos eventos de ejecución inmediata, el evento en curso se detiene y vuelve a la cola en estado *preparado*. Este evento en curso, se insertará en la cola justo después de sus eventos hijos de ejecución inmediata y volverá a estar en ejecución, una vez que éstos finalicen.

Una vez que un evento finaliza su ejecución, en función del resultado de la ejecución y del tipo de evento, el nuevo estado del evento puede ser el siguiente:

1. Si el evento ha finalizado sin éxito, el nuevo estado pasa a ser *fallido*.
2. Si el evento se ha generado con una función de temporización repetitiva (por ejemplo, utilizando la función predefinida *setInterval*), el evento vuelve a la cola y se ejecutará de nuevo, una vez transcurrido el intervalo de tiempo establecido. Su nuevo estado pasa a ser *no-preparado* (*no bloqueante*).

   Estos eventos pueden cancelarse durante la ejecución de JavaScript (por ejemplo, utilizando la función predefinida *clearInterval*). En ese caso, el evento no volverá a la cola (o se eliminará de ella).
3. Si el evento ha generado nuevos eventos, su estado pasa a ser *finalizado*, a la espera de que finalicen también, todos sus eventos hijos.

   Si sus hijos finalizan con éxito, su nuevo estado será *completado*, o en caso contrario, su nuevo estado será *fallido*, si alguno de estos eventos hijos no ha finalizado de manera satisfactoria.

   Esta comprobación es recursiva, es decir, para que finalice un evento, han de finalizar sus hijos, los hijos de éstos y así de forma sucesiva.
4. Si el evento no ha generado nuevos eventos y ha finalizado con éxito, su estado pasa a ser *completado*.

4.4.3 Manejadores de errores en la ejecución de un evento

Para manejar los posibles errores durante la ejecución de los eventos, es posible establecer políticas para configurar ante qué tipo de fallos se debe detener la ejecución de la secuencia, y qué tipo de errores pueden ser ignorados y continuar así, con la ejecución de la secuencia, procesando los eventos que quedan pendientes en la cola.

A este respecto, en la arquitectura diseñada, es posible configurar si se deben ignorar o no, los siguientes tipos de errores:

1. Fallo en la descarga de un documento HTML.
2. Fallo en la descarga de un documento externo (fichero JavaScript, hoja de estilo CSS, etc.).
3. Errores en el procesamiento de un documento HTML con estructura inválida, etiquetas cerradas de manera incorrecta, etc. (el comportamiento por defecto es este caso, es el de intentar autogenerar la estructura correcta, normalizando el documento).
4. Errores en la ejecución de un script (un nodo de tipo script incluido dentro de la página HTML, o un script disparado mediante un evento asíncrono con temporizador).
5. Errores de la ejecución de un manejador asociado a un evento, emitido sobre un nodo del árbol DOM (por ejemplo, utilizando la función JavaScript predefinida, *fireEvent*).

4.4.4 Tipos de eventos más relevantes de la arquitectura del componente de navegación

Para finalizar este apartado, se comentarán de forma breve, los tipos de eventos más relevantes, contemplados en el diseño del componente de navegación.

4.4.4.1 Eventos de navegación

Estos eventos realizan una navegación a una URL y desencadenan una serie de acciones asociadas.

Los eventos de navegación, se pueden generar tanto desde un paso de la secuencia (por ejemplo, utilizando un comando para navegar a una URL, véase el comando *Navigate* en el ejemplo de la Figura 2), como desde el subsistema de ejecución de JavaScript (por ejemplo, modificando el atributo *location* del objeto predefinido *window*).

Cuando la navegación devuelve contenido HTML, se desencadenan una serie de acciones, algunas de las cuales pueden generar nuevos eventos que se almacenarán en la cola:

1. Procesado (*parsing*) del documento HTML.
2. Descarga de los objetos externos, que han sido localizados (de forma incremental) dentro del documento (por ejemplo, ficheros JavaScript y hojas de estilo CSS).
3. Ejecución de código JavaScript (en primer lugar, se analizarán los nodos de tipo script, y durante la ejecución de éstos, se puede generar más contenido JavaScript de manera dinámica).

4.4.4.2 Eventos de ejecución de JavaScript

Se generan, sobre todo, durante el procesado del documento HTML, aunque estos eventos también se pueden generar desde la secuencia de navegación web.

Mediante un comando de la secuencia, se puede realizar una inyección de JavaScript, que en algunos casos, puede ser de utilidad para modificar el funcionamiento normal de la página web.

4.4.4.3 Eventos de simulación de acciones de usuario

Estos eventos disparan los manejadores asociados a un evento, sobre alguno de los nodos del árbol DOM de la página HTML (por ejemplo *click*, *mouseover*, *keypress*, etc.).

Los manejadores de eventos, suelen establecerse durante la ejecución inicial de los scripts contenidos en el código HTML.

Para disparar estos manejadores, se puede hacer tanto desde la secuencia de navegación (por ejemplo, con un comando para disparar un evento sobre un nodo seleccionado con anterioridad, véase el comando *FireEvent* en el ejemplo de la Figura 2), como durante la evaluación del código JavaScript (por ejemplo, utilizando la función predefinida *click* sobre alguno de los nodos del árbol DOM).

4.4.4.4 Eventos de ejecución de JavaScript con temporizadores

Estos eventos se generan, sobre todo, durante la evaluación de JavaScript, utilizando las funciones predefinidas *setTimeout* y *setInterval*.

Estas funciones establecen un tiempo de espera, transcurrido el cual, debe realizarse la ejecución del código JavaScript asociado.

Con el objetivo de permitir la ejecución de otros eventos, y no bloquear la ejecución mientras no finalice el temporizador, estos eventos mantienen un estado especial en la cola y no pasarán a estado ejecutable, hasta que no transcurra su tiempo de espera.

4.4.4.5 Eventos automáticos

Se generan de manera automática en el componente de navegación, como consecuencia de la finalización de ciertas acciones. Algunos ejemplos de eventos automáticos:

1. Evento *load*, emitido sobre el nodo *body* del árbol DOM de la página.
2. Evento *DomContentLoaded*, emitido sobre el objeto *document* cuando finaliza la carga del documento HTML, sin esperar por la carga de las hojas de estilo o subframes.

4.5 SUBSISTEMAS AUXILIARES

A continuación, se enumeran y describen en detalle, los sistemas auxiliares que componen la arquitectura del componente de navegación.

4.5.1 Subsistema optimizador

El subsistema que lleva por nombre optimizador, es un elemento fundamental dentro de la arquitectura del componente de navegación diseñado en este trabajo.

Su funcionalidad principal, es la de proporcionar los mecanismos necesarios para:

1. Generar y almacenar, durante la fase de optimización, la información que permitirá implementar algunas de las optimizaciones detalladas en capítulos anteriores.
2. Acceder a esta información de optimización, durante la fase de ejecución.

Durante la fase de optimización, este subsistema es el encargado de calcular:

1. Las dependencias entre los nodos del árbol DOM de la página HTML.
2. El grafo con las dependencias entre los nodos de tipo script.

Para realizar este cálculo de dependencias, el optimizador monitoriza todas las sentencias que ejecuta el motor de JavaScript, realizando un análisis detallado de los nodos del árbol DOM y de las variables que intervienen en cada una de estas sentencias.

Una vez calculados los fragmentos irrelevantes y el grafo de dependencias entre scripts, este componente permite identificar a los nodos involucrados, por medio de expresiones XPath.

Para ello, en este subsistema deben implementarse los algoritmos que permiten identificar a un nodo, de forma única, dentro del árbol DOM (algoritmos detallados en el apartado 3.1.3).

El componente optimizador, también es el encargado de almacenar de forma persistente, todas las expresiones generadas, junto con la información que permite identificar a los documentos sobre los que se pueden aplicar estas expresiones (los algoritmos de identificación de documentos se han explicado en el apartado 3.1.4).

Durante la fase de ejecución, el subsistema que procesa los documentos HTML (durante la construcción del árbol DOM de la página web), interactuará con el optimizador para detectar los fragmentos irrelevantes, preguntándole, en primer lugar, si tiene información de optimización asociada a la página que se va a cargar.

En caso afirmativo, antes de añadir cada nodo al árbol DOM de la página, el componente optimizador determina si ese nodo es irrelevante. Los nodos detectados como tal, se descartan de forma automática y no se añaden al árbol.

De manera adicional, durante la fase de ejecución, el *thread* que despacha los scripts para ser ejecutados en los *threads* auxiliares del pool, utiliza este

componente optimizador para detectar los scripts que se pueden evaluar de manera concurrente.

Cuando el motor de ejecución de JavaScript finaliza la evaluación de un script, el optimizador actualiza el grafo de dependencias y realiza una notificación al *thread* despachador. En ese momento, si aparecen nuevos scripts preparados para ejecutarse, éstos serán despachados en alguno de los *threads* libres del pool (o en el *thread* principal, si se trata del primer evento en la cola y este *thread* queda libre).

4.5.2 Subsistema de procesamiento de HTML y XML

El subsistema de procesamiento (o *parsing*) de documentos HTML y XML, es el encargado de analizar los flujos que contienen datos HTML y XML, con el objetivo de generar el árbol DOM de esos documentos.

Para maximizar el rendimiento, este componente utiliza un procesador de tipo SAX, que permite descubrir elementos relevantes de manera incremental (por ejemplo, nodos de tipo script u hojas de estilo CSS). Esto permite desencadenar acciones de pre-procesamiento durante la generación del documento HTML.

Por ejemplo, la descarga de hojas de estilo CSS se inicia de forma inmediata, una vez que éstas son identificadas dentro la página (en paralelo con la construcción del árbol DOM).

Para descargar varios documentos de forma paralela, el subsistema de red dispondrá de un pool de *threads*, dedicado en exclusiva a esta tarea.

Este procesador de datos HTML y XML, se utiliza dentro del componente de navegación, en diferentes tipos de eventos:

1. Para construir el árbol DOM a partir de un documento HTML descargado de la Web.
2. Para procesar peticiones AJAX que devuelven contenido XML o HTML.
3. Para procesar de manera incremental fragmentos de código HTML generados durante la evaluación del código JavaScript.

   Por ejemplo, fragmentos de datos HTML generados por medio de la función JavaScript *write,* predefinida en el objeto *document*, o utilizando la propiedad *innerHTML* de los elementos del árbol DOM.

Durante la fase de optimización, cada vez que se realiza la descarga de una página HTML, el subsistema procesador de HTML/XML, utiliza el optimizador para detectar los fragmentos irrelevantes y descartarlos del árbol DOM que se está construyendo.

Para ello, en primer lugar, se consulta si existe información de optimización asociada a ese documento.

En caso de existir esta información, durante el proceso de construcción del árbol DOM, se consulta al optimizador antes de añadir cada nodo al árbol, para determinar si ese nodo se corresponde con alguna de las expresiones XPath

almacenadas y que representan a los subárboles detectados como irrelevantes, durante la fase de optimización.

Los nodos detectados como irrelevantes, se descartan de forma automática y no se añaden al árbol.

4.5.3 Subsistema CSS

El subsistema CSS, es el encargado de procesar y almacenar las hojas de estilo CSS.

Cada hoja de estilo, estará formada por una serie de reglas con atributos de presentación, que serán aplicadas sobre distintos nodos del documento HTML.

Durante la generación del árbol DOM, el procesador de HTML detecta los nodos de tipo CSS, tanto aquellos que residen en ficheros externos y que serán descargados con anterioridad por la capa de red, como aquellos que se encuentran incrustados, dentro del código HTML.

Todos estos elementos se envían al subsistema CSS para que los procese y para que almacene las diferentes reglas contenidas en ellos.

El subsistema CSS, será también el encargado de aplicar estas reglas sobre aquellos nodos del árbol DOM, cuyos atributos visuales sean consultados, durante la evaluación del código JavaScript.

Para aquellos nodos, para los cuales su información de visualización no sea consultada durante la ejecución de JavaScript, los atributos de presentación no se calculan, lo que contribuye a mejorar la eficiencia del sistema.

Además, el componente de navegación permite deshabilitar de forma manual, ciertas funcionalidades del subsistema CSS, por medio de parámetros de configuración.

Esta configuración manual, complementa a la técnica de optimización automática de evaluación de estilos CSS bajo demanda y puede ser realizada por usuarios avanzados.

Así pues, se pueden establecer los siguientes cambios en el comportamiento del subsistema CSS del motor de renderización, durante la construcción del árbol DOM:

1. Los documentos CSS externos no se descargan, con lo que el uso de la red se minimiza.
2. Los documentos CSS incrustados dentro del propio documento HTML no son procesados, con lo que el uso de CPU se minimiza.
3. Durante la ejecución de código JavaScript, se devuelven valores predefinidos para cada uno de los atributos de visualización.

Es necesario destacar que la técnica de optimización automática, al no utilizar ningún tipo de información previa, no permite conocer de antemano si una hoja de estilos CSS se va a utilizar o no durante la evaluación de JavaScript, por lo tanto, para obtener un grado mayor de eficiencia, es necesario combinar la técnica de

optimización automática con la configuración manual para desactivar manualmente la descarga de hojas de estilo.

4.5.4 Motor de ejecución de JavaScript

El motor de ejecución de JavaScript, es el subsistema encargado de la ejecución de los scripts contenidos en una página web y también de aquellos otros scripts que se generan de forma dinámica, durante la evaluación de los primeros.

Además de la ejecución del código JavaScript, este subsistema también es el encargado de compilar y almacenar los scripts, en formato binario, dentro de una caché persistente, habilitada para tal efecto.

En la mayoría de los casos, esta caché con los scripts compilados aumenta de manera importante la eficiencia del sistema, dado que el tiempo de *parsing*, supone un porcentaje considerable, dentro de todo el proceso de evaluación.

Durante la fase de optimización, el motor de ejecución de scripts hace uso del subsistema optimizador para calcular las dependencias entre los diferentes nodos del árbol DOM de la página.

Con posterioridad, una vez que ha finalizado la evaluación de todos los scripts y cuando la carga de la página se ha completado, el optimizador analiza las dependencias obtenidas durante la ejecución y genera la información de optimización con los fragmentos irrelevantes y con el grafo de dependencias entre scripts.

De manera adicional, el motor de ejecución de JavaScript permite una configuración de más bajo nivel, de grano fino, especialmente indicada para usuarios avanzados, con un alto conocimiento sobre las páginas web a las que se accede durante la ejecución de una secuencia de navegación web determinada.

La configuración manual, permite mejorar el rendimiento de aquellos scripts detectados como relevantes y necesarios para la ejecución de la secuencia, pero que aun siendo relevantes, contienen acciones no necesarias (cabe destacar que la técnica de optimización automática permite descartar el script completo, no un fragmento de éste).

Por lo tanto, la mayoría de estos parámetros, son compatibles con las técnicas de optimización automática, con lo que podrían combinarse ambos, en la ejecución de una misma secuencia.

Entre otras cosas, la configuración de grano fino del motor de JavaScript permite:

1. Deshabilitar de forma manual algunas funciones predefinidas, como los temporizadores *setTimeout* y *setInterval*.

   En muchas ocasiones, el código ejecutado mediante estas funciones es irrelevante para el correcto funcionamiento de la secuencia de navegación web. Por ejemplo, es frecuente que estas funciones se utilicen para cargar nuevos anuncios de publicidad cada cierto tiempo.

En estos casos, el comportamiento del componente de navegación, se puede variar mediante parámetros de configuración para descartar los eventos generados con estas funciones temporizador.

2. Algunos objetos predefinidos, se pueden configurar para prevenir la ejecución de acciones innecesarias.

   Por ejemplo, el objeto predefinido *XmlHttpRequest* ejecuta peticiones AJAX, que en muchos casos solicitan información irrelevante (por ejemplo, contadores de visitas, etc.).

   Este objeto se puede configurar para que no ejecute la petición HTTP y para que devuelva valores predefinidos, tanto para el contenido de la respuesta, como para el código de estado.

3. Algunas funcionalidades avanzadas de JavaScript, se pueden deshabilitar para prevenir cálculos innecesarios.

   Por ejemplo, el objeto predefinido *document* mantiene un índice con todos los elementos en el DOM que tienen atributo *id* o *name*. Si el código script no utiliza este método de acceso para obtener las referencias a los nodos por atributo *id* o *name*, esta estructura no sería necesaria y se podría deshabilitar de manera manual.

Los parámetros de configuración manual, se pueden modificar de forma individual, en cada uno de los navegadores y también podrían modificarse de manera dinámica, en cualquier punto de la ejecución de la secuencia (véase el Anexo I, en el que se detalla cómo funciona el mecanismo de configuración manual del componente de navegación).

El motor de JavaScript, puede deshabilitarse en su totalidad, también por medio de parámetros de configuración. En este caso, la creación de objetos sin soporte de JavaScript será más rápida, dado que se evita la creación de algunas estructuras internas propias de los objetos con soporte JavaScript.

4.5.5 Subsistema de red

El subsistema de red es el encargado de las llamadas de red, como por ejemplo, las peticiones HTTP que realizan la descarga de documentos de la Web.

Este subsistema, también se encarga de otro tipo de peticiones, como por ejemplo, las peticiones AJAX generadas durante la evaluación de JavaScript, utilizando el objeto *XmlHttpRequest* [XMLHTTP].

Para la descarga de documentos de la Web, el subsistema de red mantiene su propio pool de *threads*. Esto permite la descarga en paralelo de múltiples elementos.

Este pool permite establecer un número máximo de descargas en paralelo y también permite establecer un número máximo de descargas por cada sitio web al que se accede.

Durante la construcción del árbol DOM de la página, a medida que el procesador HTML va descubriendo elementos descargables (sobre todo, scripts y hojas de estilo), estos objetos se envían de manera inmediata al subsistema de red. Éste, es el encargado de descargarlos utilizando el pool de *threads* dedicado, utilizando una cola de descargas pendientes, en la que se almacenan los objetos, cuando se ha alcanzado el máximo número de descargas (globales o por sitio web).

Una vez que finaliza la descarga, es cuando la acción asociada al objeto descargable se puede llevar a cabo. Por ejemplo, la ejecución de JavaScript, cuando se trata de un nodo de tipo script, o el procesado de la hoja de estilo, cuando se trata de un nodo de tipo *LINK*.

La capa de red, también soporta otras funcionalidades más avanzadas, mediante el establecimiento de parámetros de configuración. Estas funcionalidades, también están orientadas a usuarios avanzados con un alto conocimiento de bajo nivel de los sitios web a los que se accede:

1. Bloqueo de las descargas que se realizan sobre un conjunto de sitios web predefinidos.

   Esta lista de sitios bloqueados, se puede configurar mediante parámetros y sobre todo, incluye servidores que ofrecen contenido que no es relevante para la ejecución de la secuencia:

   a) Sitios web de publicidad.

   b) Sitios web de rastreo y contadores de visitas.

   c) Sitios web de *plugins* de redes sociales.

2. Modificación del valor de diversas cabeceras de la petición HTTP, por ejemplo, la cabecera *User-Agent* que permite al servidor identificar qué tipo de navegador se está conectando.

   Esta funcionalidad puede ser útil cuando el servidor ofrece diferentes vistas, en función del navegador que se conecta a él. Es frecuente que muchos sitios web ofrezcan una vista simplificada para dispositivos móviles, que ofrece la misma información, pero que reduce el consumo de recursos.

4.5.6 Subsistema de persistencia de datos

El subsistema de persistencia de datos, ofrece un mecanismo para almacenar y recuperar aquella información persistente en el componente de navegación.

Entre la información persistente que utiliza el componente de navegación, se pueden destacar los siguientes elementos:

1. Caché de documentos descargados: incluye, sobre todo, documentos JavaScript y hojas de estilo CSS.

2. Caché con código JavaScript compilado (en formato binario): incluye, tanto scripts almacenados en documentos externos, como scripts insertados dentro del propio documento HTML.

3. Cookies persistentes.

   De manera adicional, en el diseño de este subsistema, se incluye un adaptador que permite obtener las cookies almacenadas por otros navegadores (por ejemplo, las de Microsoft Internet Explorer), en la cuenta de usuario en la que se está ejecutando el componente de navegación.

   Al acceder a cookies persistentes almacenadas en la cuenta del usuario, en algunos casos, evita el proceso de identificación al acceder a determinados sitios web.

4. Información de optimización: incluye tanto las expresiones XPath que identifican a los fragmentos irrelevantes, como los grafos con las dependencias entre scripts.

5. Valores por defecto de los parámetros de configuración del componente de navegación.

Este subsistema, ofrece un API genérico de alto nivel, que permite que cualquier implementación de la arquitectura desarrolle el mecanismo concreto para establecer la persistencia de los datos (por ejemplo, mediante una base de datos, mediante un fichero de propiedades de configuración, etc.), sin que ello afecte al resto de subsistemas.

4.6 OBJETOS DEL COMPONENTE DE NAVEGACIÓN

Todos los subsistemas definidos en los apartados anteriores, pueden acceder a los objetos del navegador.

Estos objetos, están almacenados en memoria e incluyen, entre otros, las ventanas y los documentos que se van generando durante la ejecución de la secuencia de navegación.

La Figura 64, muestra los objetos más relevantes.

Cada ventana contendrá el documento actual y otra serie de objetos, a algunos de los cuales, se podrá acceder durante la evaluación del código JavaScript (por ejemplo, *navigator* e *history*).

En un primer momento, el componente de navegación, se crea con una sola ventana que contiene un documento vacío.

Tras la primera navegación, una vez que se ha construido el documento HTML, éste pasa a ser el documento actual de la ventana. Cada vez que se carga un nuevo documento, se reemplaza el documento actual y el anterior se almacena en el histórico (utilizando el objeto *history*).

Al histórico, con los documentos cargados con anterioridad en la misma ventana, también se podrá acceder desde el contexto de ejecución de JavaScript.

El tamaño máximo de documentos a almacenar en el histórico, se puede configurar e incluso puede ser cero, lo que implica que el histórico se deshabilita. Esta configuración, puede ser útil en sitios web que no acceden a este objeto predefinido desde JavaScript, mejorando en estos casos, el uso de memoria (en cada ventana, sólo se almacenará el último de los documentos cargados, y los anteriores se descartan).

Durante la ejecución de la secuencia de navegación, se pueden crear ventanas adicionales, por ejemplo, utilizando desde JavaScript la función predefinida *open*, del objeto *window*.

Se puede acceder a todas las ventanas, así como también a los marcos (*frames* o *iframes*), desde cualquier subsistema. Para ello, es necesario utilizar el objeto gestor de ventanas (*Window Manager*).

Este objeto también permite establecer una configuración avanzada, mediante la modificación de ciertos parámetros. Por ejemplo, es habitual que muchos sitios web utilicen componentes *iframe*, para incluir publicidad que no es relevante a la hora de ejecutar la secuencia de navegación. En estos casos, el gestor de ventanas se puede configurar para que active un bloqueador de ventanas y marcos, que evita la carga y procesamiento de estos elementos en los sitios web indicados.

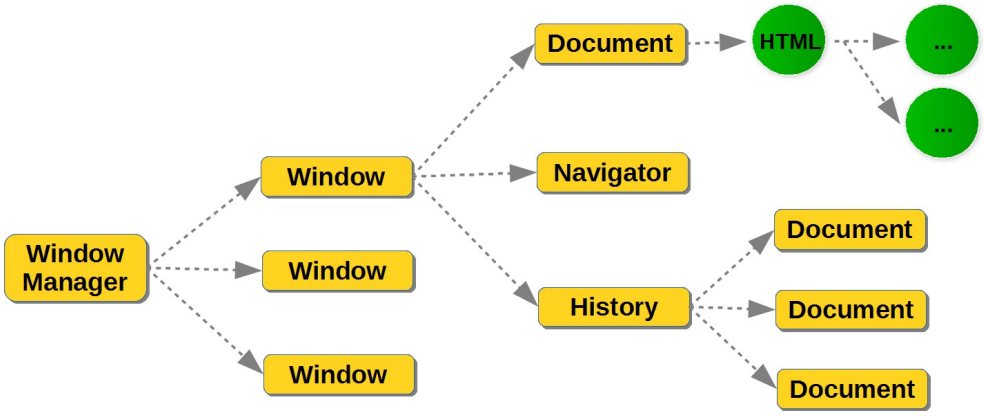


Figura 64. Objetos del componente de navegación.

Para permitir la ejecución en paralelo del código JavaScript, algunas estructuras internas de los objetos del navegador han de estar sincronizadas, para prevenir errores de acceso concurrente. Algunas de estas estructuras sincronizadas son las siguientes:

1. La lista de hijos de cualquier elemento del árbol DOM.
2. Algunas listas predefinidas en el objeto *document*:
    a) Lista de enlaces: *document.anchors*.
    b) Lista de formularios: *document.forms*.
    c) Lista con todos los elementos: *document.all*.
3. La lista de entradas (nodos de tipo *input*, *select* y *textarea*) de un formulario.
4. La lista con las opciones (nodos de tipo *option*) de un nodo de tipo selección.

4.7 CONFIGURACIÓN MANUAL DEL COMPONENTE DE NAVEGACIÓN

Además de dar soporte a las optimizaciones automáticas comentadas con anterioridad, el componente de navegación proporciona un mecanismo de configuración manual, que permite que determinados usuarios avanzados, puedan variar diferentes aspectos en el funcionamiento del navegador, basándose en las características de bajo nivel de cada uno de los sitios web a los que se accede, todo ello con el objetivo de mejorar la eficiencia en la ejecución de la secuencia.

El mecanismo de configuración manual del componente de navegación, es compatible con las técnicas de optimización automática desarrolladas con anterioridad, pudiendo combinarse ambas en la ejecución de la misma secuencia.

En el Anexo I, se describe en detalle, cómo funciona el mecanismo de configuración del componente de navegación, así como la lista con los parámetros de configuración más relevantes.

4.8 ETAPAS DE PROCESAMIENTO DEL MOTOR DE RENDERIZACIÓN

La Figura 65, muestra las etapas de procesamiento que utiliza un componente de navegación, construido conforme a la arquitectura presentada en los apartados anteriores, y que da soporte a las diferentes técnicas de optimización, detalladas en el capítulo 3:

1. Evaluación de estilos CSS bajo demanda, sobre un subconjunto de nodos reducido (aquellos, cuyos atributos de presentación son accedidos durante el proceso de evaluación de JavaScript).

2. Ejecución de JavaScript en paralelo, cuando no existen dependencias entre los nodos de tipo script contenidos en el documento HTML.

3. Construcción de un árbol DOM minimizado, descartando los fragmentos de la página que no son necesarios para la correcta ejecución de la secuencia de navegación web.

Las diferencias entre el procesamiento que se realiza en el motor de renderización de un navegador convencional (véase la Figura 10, con las etapas de procesamiento de los navegadores convencionales), comparado con el procesamiento que se realiza en el componente de navegación diseñado siguiendo la arquitectura propuesta, se pueden resumir en los siguientes puntos:

1. El navegador a medida no necesita interfaz gráfica de usuario, con lo que algunos de los pasos de generación de información visual y renderización, no son necesarios.

2. Con la evaluación de estilos CSS bajo demanda, se minimizan los cálculos a realizar, para generar la información de visualización en los nodos del árbol DOM.

3. Durante el proceso de construcción del árbol DOM, se identifican y descartan los fragmentos irrelevantes en la página. Como consecuencia de esto, se construye un árbol DOM mucho más pequeño, minimizando el uso de CPU, memoria y ancho de banda de la red.

4. La evaluación de JavaScript puede ejecutarse en paralelo, cuando los scripts no tienen dependencias entre ellos. Con esta optimización, el tiempo de evaluación de JavaScript se reduce.

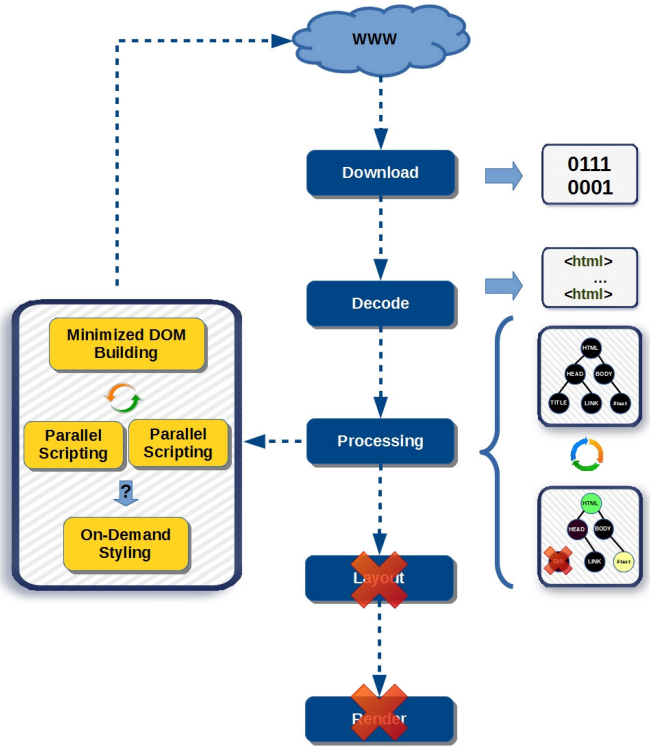


Figura 65. Esquema del motor de renderización.

5 EXPERIENCIA OBTENIDA CON EL USO DEL SISTEMA

En este capítulo, se valida, por un lado, la eficacia de las distintas técnicas de optimización propuestas, y por otro lado, el rendimiento de la arquitectura para un componente de navegación a medida, diseñada en este trabajo.

La estructura del capítulo es la siguiente. En el apartado 5.1, se detallan los experimentos realizados sobre la implementación de referencia de un navegador a medida, que sigue las especificaciones de la arquitectura propuesta en el capítulo 4 y que soporta las diferentes técnicas y algoritmos de optimización, presentados en el capítulo 3.

En el apartado 5.2, se comenta de manera breve, la utilización en aplicaciones industriales de la implementación de referencia de la arquitectura diseñada en esta tesis doctoral.

En último lugar, en el apartado 5.3, se realiza un análisis del cumplimiento de los objetivos, expuestos en el apartado 1.2, a la luz de lo descrito en los apartados anteriores.

5.1 EXPERIMENTOS

Para validar la arquitectura de navegación propuesta en este trabajo, se ha desarrollado un sistema de navegación a medida que sigue las especificaciones de dicha arquitectura y que, por lo tanto, implementa las diferentes técnicas y algoritmos de optimización detallados en los capítulos anteriores.

En los siguientes apartados, se detallan las pruebas realizadas para validar dicha arquitectura.

En primer lugar, en el apartado 5.1.1, se describen las principales características de la implementación de referencia del componente de navegación.

A continuación, se detallan los experimentos diseñados para medir la eficiencia de cada una de las técnicas de optimización por separado, así como las pruebas realizadas para medir la eficiencia del sistema completo, comparándolo con otros componentes de navegación:

1. En el apartado 5.1.2, se muestran las pruebas realizadas para medir la eficiencia de la técnica de construcción de un árbol DOM minimizado.
2. A continuación, en el apartado 5.1.3, se muestran las pruebas realizadas para medir la eficiencia de la técnica de ejecución de JavaScript en paralelo.
3. En el apartado 5.1.4, se muestran las pruebas realizadas para medir la eficiencia de la técnica de evaluación de estilos CSS bajo demanda.
4. Por último, en el apartado 5.1.5, se detallan las pruebas realizadas para medir la eficiencia de toda la arquitectura, comparando la implementación de referencia con otros sistemas de navegación.

En los experimentos realizados, se han seleccionado una serie de sitios web de diferentes países y de diferentes dominios, todos ellos incluidos en la lista publicada por Alexa [ALEXA], con los sitios web más visitados.

5.1.1 Implementación de la arquitectura en un componente de navegación

Para validar la arquitectura desarrollada en este trabajo, se ha realizado una implementación de referencia, que sigue todos los planteamientos y especificaciones de dicha arquitectura.

Por un lado, el componente de navegación implementa todos los subsistemas detallados en el capítulo 4 y por otro lado, implementa también todos los algoritmos detallados en el capítulo 3, con el objetivo de dar soporte a las diferentes técnicas de optimización.

La implementación de referencia de la arquitectura, se compone de un sistema de navegación a medida, que emula al popular navegador convencional Microsoft Internet Explorer.

La versión de Microsoft Internet Explorer que simula, se puede configurar desde la más antigua soportada, la versión 8, hasta la más reciente, la versión 11 (esta última, da soporte a un mayor número de estándares, con lo que el modo de funcionamiento se parece más al de otros navegadores convencionales).

La implementación de referencia, ha sido desarrollada en el lenguaje de programación Java y se han utilizado una serie de librerías de código abierto, para implementar algunas funcionalidades relevantes dentro del sistema de navegación.

Entre estas librerías de libre distribución, se pueden destacar las siguientes:

1. *Mozilla Rhino* [MOZRHN]: librería desarrollada por la corporación Mozilla y que permite interpretar y evaluar código JavaScript.

   Además, para dar soporte a las técnicas de construcción de un árbol DOM minimizado y de evaluación de JavaScript en paralelo, sobre esta librería se han realizado diversas modificaciones, que permiten implementar el cálculo de las dependencias entre los nodos del árbol DOM durante la ejecución del código JavaScript.

2. *CyberNeko HTML Parser* [NEHTML]: librería de código abierto, utilizada para el procesamiento de documentos HTML.

   Esta librería proporciona diferentes tipos de analizadores, entre ellos el procesador de tipo SAX, que se ha utilizado para generar los documentos a partir del código fuente HTML.

3. *Apache Commons Http Client* [APHTTPC3]: librería de código abierto, desarrollada por la fundación Apache y que ofrece un API de alto nivel, para emitir las peticiones HTTP generadas durante la ejecución de la secuencia de navegación web.

Esta librería da soporte a diversas funcionalidades, como la gestión de cookies, utilización de HTTP seguro, etc.

4. Se han utilizado otros proyectos de código abierto, entre los que se pueden destacar las diferentes librerías utilidad, desarrolladas por la fundación Apache [APCOMM]:

    a) *Apache Commons Pool*: utilizada para implementar diferentes tipos de pools de objetos (por ejemplo, el pool de *threads* reutilizables, dedicados a evaluar JavaScript en paralelo).

    b) *Apache Commons Logging*: utilizada para generar mensajes de depuración, que permite trazar y depurar la ejecución de la secuencia de navegación web.

    c) *Apache Commons Lang*: librería que contiene utilidades para manipular diferentes tipos de elementos del lenguaje Java, por ejemplo, utilidades para la manipulación de cadenas de caracteres.

    d) *Apache Commons IO*: librería con distintas utilidades para la lectura y manipulación de flujos de entrada y salida (por ejemplo, ficheros, sockets, etc.).

La implementación de referencia no tiene interfaz gráfica, ni capacidades de renderización, pero tiene soporte para calcular las reglas almacenadas en hojas de estilo CSS, y aplicarlas sobre los nodos del árbol DOM.

En esta implementación, también se da soporte a otras funcionalidades disponibles en los navegadores comerciales, como Cookies, peticiones AJAX asíncronas, soporte de la práctica totalidad de los objetos y funciones JavaScript predefinidos en los navegadores convencionales, etc.

Algunas funcionalidades exclusivas del navegador Internet Explorer, no están soportadas en la implementación de referencia, como por ejemplo, la evaluación de lenguajes de scripting propietarios (como VBScript), o el soporte para objetos incrustados (por ejemplo Adobe Flash).

En los siguientes apartados, se detallan las pruebas realizadas sobre esta implementación de referencia, para medir la eficiencia de la arquitectura del componente de navegación.

En primer lugar, se muestran las pruebas realizadas para evaluar la eficiencia de cada una de las técnicas de optimización por separado (evaluación de estilos CSS bajo demanda, construcción de un árbol DOM minimizado y evaluación de JavaScript en paralelo) y, a continuación, se detallan las pruebas que evalúan la eficiencia de toda la arquitectura, comparando la implementación de referencia con otros componentes de navegación, tanto navegadores desarrollados a medida como componentes que utilizan el API de alto nivel de algún navegador convencional.

5.1.2 Evaluación de la técnica de construcción de un árbol DOM minimizado

En este apartado, se muestran los experimentos realizados para medir la eficiencia de la técnica de optimización, que permite la construcción de un árbol DOM minimizado de la página HTML, tras descartar los fragmentos detectados como irrelevantes durante la fase de optimización.

El primer experimento realizado en este apartado, compara los recursos consumidos por la implementación de referencia del componente de navegación cuando opera en modo normal, emulando a un navegador convencional y cargando las páginas en su integridad, con los recursos consumidos por la implementación de referencia del componente de navegación, cuando utiliza sus capacidades de optimización, para detectar y construir un árbol DOM minimizado, descartando los fragmentos irrelevantes.

En este experimento, se ha seleccionado un conjunto de sitios web reales, todos ellos incluidos en la lista de Alexa de sitios web más visitados en todo el mundo.

Sobre cada uno de estos sitios web, se ha generado una secuencia de navegación que atraviesa varias páginas, o bien siguiendo enlaces, o bien enviando formularios (una vez que éstos han sido rellenados con los valores representativos).

En primer lugar, se ha realizado una primera ejecución de cada secuencia de navegación. En esta primera ejecución de la secuencia, se ha calculado y almacenado la información de optimización, con los fragmentos irrelevantes detectados.

A continuación, se han comparado los recursos consumidos por el componente de navegación cuando utiliza la información de optimización, generada con anterioridad, con una ejecución normal de la secuencia, que no utiliza las capacidades de optimización.

Para prevenir problemas derivados de pequeñas variaciones en una misma página web, cuando se accede a ella en momentos diferentes, cada secuencia de navegación se ha ejecutado 10 veces y los resultados que se muestran, representan el valor medio de las 10 ejecuciones.

	IRRELEVANT NODES PER PAGE	XPATH-LIKE EXPRESSIONS LENGTH	TIME CALCULATING NODE DEPENDENCIES	TIME GENERATING XPATH-LIKE EXPRESSIONS	TOTAL EXECUTION TIME
ALEXA	43,5	1,2	36 (0,81%)	95 (2,15%)	4426
AMAZON	78,25	1,14	46 (0,54%)	364 (4,26%)	8549
APPLESTORE	57,67	1,6	61 (1,44%)	130 (3,07%)	4228
BARNES&NOBLE	49	1,19	54 (0,75%)	149 (2,07%)	7187
BLOOMBERG	66,67	1,29	56 (0,71%)	322 (4,07%)	7908
CNET	71,67	1,23	92 (0,8%)	177 (1,53%)	11563
CNN	37,33	1,46	72 (0,77%)	218 (2,35%)	9294
EBAY	159,25	1,38	39 (0,47%)	693 (8,27%)	8377
FLICKR	30,25	1,39	70 (0,75%)	67 (0,72%)	9338
GOOGLENEWS	47,25	1,38	42 (0,72%)	187 (3,22%)	5810
IMDB	120,33	1,35	79 (0,84%)	268 (2,86%)	9361
LINKEDIN	86,33	1,92	43 (0,69%)	252 (4,04%)	6230
REFERENCE	152,5	1,28	95 (0,75%)	189 (1,5%)	12639
REUTERS	52,75	1,62	99 (0,51%)	212 (1,1%)	19341
SOFTONIC	59,75	1,27	38 (0,58%)	184 (2,8%)	6579
SPIEGEL	123,5	1,48	48 (0,5%)	376 (3,93%)	9570
STACKOVERFLOW	34,67	1,3	56 (0,83%)	121 (1,79%)	6770
TARINGA	72,67	1,25	65 (0,47%)	126 (0,92%)	13746
THEGUARDIAN	106,33	1,18	124 (1,01%)	383 (3,13%)	12219
TRIPADVISOR	41,75	1,18	15 (0,3%)	307 (6,24%)	4921
W3CSCHOOLS	53	1,36	51 (0,63%)	89 (1,09%)	8143
WALMART	97	1,26	73 (0,58%)	531 (4,23%)	12554
WIKIPEDIA	51	1,67	81 (1,13%)	249 (3,46%)	7192
WORDPRESS	29,5	1,17	37 (0,64%)	41 (0,71%)	5776
WSJOURNAL	90	1,55	149 (0,71%)	229 (1,09%)	21028
YAHOO	77,5	1,06	45 (0,51%)	107 (1,21%)	8875
YELP	76	1,13	41 (0,61%)	176 (2,62%)	6706
GLOBAL	72,79	1,34	1707 (0,69%)	6242 (2,51%)	248330

Tabla 1. Tiempo de generación de fragmentos XPath.

La Tabla 1, muestra las siguientes mediciones para cada uno de los sitios web:

1. Número medio de expresiones XPath generadas por cada página, a la que se accede durante la ejecución de la secuencia (o lo que es lo mismo, número medio de fragmentos o nodos irrelevantes, detectados en cada página).

2. Longitud media de las expresiones generadas (o lo que es lo mismo, número medio de nodos utilizados, de media, en cada expresión XPath).

3. Tiempo total (en milisegundos) consumido en el cálculo de las dependencias entre los nodos durante la ejecución del código JavaScript (y entre paréntesis, el porcentaje que supone con respecto al tiempo total de una ejecución normal de la secuencia de navegación web).

4. Tiempo total (en milisegundos) consumido en el cálculo de los fragmentos irrelevantes y los eventos automáticos necesarios, más el tiempo necesario para generar y almacenar, de manera persistente, las expresiones XPath (y entre paréntesis, el porcentaje que supone con respecto al tiempo total de una ejecución normal de la secuencia de navegación web).

5. Tiempo total consumido por una ejecución normal de la secuencia de navegación web (en milisegundos).

El número de expresiones XPath generadas por página, varía entre 29,5 y 159,25 con una media global de 72,79 expresiones por página.

Como se mostrará más adelante, este número es muy inferior al número total de nodos de cada una de las páginas (creados en una ejecución normal de la secuencia de navegación) y también, es muy inferior al número total de nodos descartados tras la construcción del árbol minimizado.

La longitud media de las expresiones XPath generadas, siempre es mayor que 1, lo que implica que en todos los sitios web, existen nodos que no se pueden identificar de manera única por sí mismos en la página HTML y es necesario utilizar más de un nodo en cada una de las expresiones generadas.

Sin embargo, esta longitud es menor que 2 en todos los casos, con lo que el número de nodos involucrados en la generación de cada uno de los fragmentos puede considerarse reducido, y por lo tanto, las expresiones generadas serán resistentes ante pequeños cambios en el código fuente de las páginas web a las que se ha accedido.

Por último, en la tabla, se puede observar como el tiempo consumido en el cálculo de las dependencias entre los nodos, y en la generación de las expresiones XPath, es muy pequeño (representa el 0,69% y el 2,51% del tiempo total de una ejecución normal de la secuencia).

Por lo tanto, se puede concluir que el proceso de generación de la información de optimización, para identificar a los fragmentos irrelevantes, es muy eficiente, y por lo tanto, este proceso se puede ejecutar con frecuencia, para regenerar la información de optimización, en escenarios en los que las páginas a las que se accede presentan cambios muy acusados y muy frecuentes.

La Tabla 2, muestra una comparativa de los recursos consumidos por una ejecución normal de la secuencia de navegación web, con respecto a los recursos consumidos por una ejecución optimizada de la misma secuencia de navegación web.

En esta tabla, cada una de las celdas muestra, en primer lugar, el valor en la ejecución normal, seguido por el valor en la ejecución optimizada.

	HTML DOM NODES CREATED	SCRIPTS EXECUTED	FRAMES AND WINDOWS	HTML PAGES DOWNLOADED	EXTERNAL OBJECTS DOWNLOADED	AJAX REQUESTS
ALEXA	1176/144	48/20	1/1	2/2	27/16	0/0
AMAZON	7965/4047	176/77	6/2	9/4	13/5	2/1
APPLESTORE	2611/79	69/1	1/1	3/3	15/11	2/0
BARNES&NOBLE	3989/136	26/26	1/1	4/4	14/14	0/0
BLOOMBERG	6281/187	243/28	14/11	8/7	53/6	0/0
CNET	3395/157	113/56	7/4	9/6	52/24	0/0
CNN	4539/40	103/8	6/1	7/3	30/5	0/0
EBAY	4932/3175	80/37	4/1	8/4	25/9	0/0
FLICKR	1332/61	61/9	2/1	5/4	19/1	0/0
GOOGLENEWS	7460/114	48/11	2/1	4/4	9/3	0/0
IMDB	2608/485	183/56	28/1	8/3	34/10	4/3
LINKEDIN	2095/167	52/12	3/1	5/3	20/5	3/0
REFERENCE	2709/579	152/29	7/2	9/3	33/11	0/0
REUTERS	2797/298	265/50	11/2	12/3	156/41	4/1
SOFTONIC	4932/250	79/6	12/1	15/4	17/3	0/0
SPIEGEL	3361/139	92/25	20/3	21/4	22/7	1/0
STACKOVERFLOW	3950/153	43/9	1/1	3/3	21/5	4/1
TARINGA	2530/256	209/15	10/1	13/3	47/8	7/0
THEGUARDIAN	4519/248	257/70	5/1	4/3	76/28	0/0
TRIPADVISOR	6769/88	92/14	1/1	4/4	6/0	0/0
W3CSCHOOLS	2380/32	89/0	8/1	8/3	33/0	0/0
WALMART	6926/385	208/29	15/3	4/3	42/13	4/3
WIKIPEDIA	5078/143	52/24	1/1	4/4	37/21	0/0
WORDPRESS	472/37	56/21	6/1	7/2	18/10	0/0
WSJOURNAL	6303/1148	204/118	39/24	60/35	78/50	0/0
YAHOO	1946/85	127/33	7/1	8/2	16/2	0/0
YELP	2815/508	52/6	7/1	2/2	14/0	0/0
TOTAL	**105870/13141 (12,41%)**	**3179/790 (24,85%)**	**225/70 (31,11%)**	**246/125 (50,81%)**	**927/308 (33,23%)**	**31/9 (29,03%)**

Tabla 2. Comparación de recursos consumidos en una ejecución normal y una optimizada.

En la tabla, se pueden observar las siguientes mediciones:

1. Número total de nodos creados.
2. Número total de nodos script creados y ejecutados.
3. Número total de ventanas y marcos creados.
4. Número total de descargas de documentos HTML realizadas (es necesario destacar que no todos los marcos o ventanas involucran la descarga de documentos HTML, algunos de estos elementos se generan mediante JavaScript, sin necesidad de realizar nuevas descargas).
5. Número total de descargas de objetos externos (sobre todo, ficheros con contenido JavaScript y hojas de estilo CSS).
6. Número total de peticiones AJAX realizadas.

Comparando los recursos consumidos en ambas ejecuciones, se puede observar como la ejecución optimizada sólo necesita crear el 12,41% de los nodos.

Al descartar ese gran volumen de nodos, también se evita un alto porcentaje de las descargas asociadas a nodos descartados, así como la evaluación de scripts no necesarios.

Más en detalle, en la ejecución optimizada, tan solo se realizan las siguientes operaciones:

- Sólo se ejecutan el 24,85% de los scripts.
- Sólo se crean el 31,11% de los marcos y ventanas.
- El número de descargas de documentos HTML se reduce a la mitad, realizándose tan solo el 50,81% de ellas.
- El número de descargas de elementos externos se reduce al 33,23%.
- Y la ejecución de peticiones AJAX es de tan solo el 29,03%.

Por lo tanto, en la ejecución optimizada se minimizará de manera considerable el uso de CPU, de memoria y de ancho de banda.

A continuación, en la Tabla 3, se muestra cómo se traduce esta minimización en el número de recursos consumidos, en una mejora en el tiempo de ejecución de la secuencia de navegación.

En esta tabla, se muestran los tiempos de ejecución (en milisegundos) de diferentes tareas esenciales en el proceso de ejecución de la secuencia de navegación. Estas tareas son las siguientes:

1. Construcción del árbol DOM de la página (incluyendo la creación de marcos y ventanas cuando sea necesario).
2. Ejecución de los scripts incluidos en la página HTML, así como otros scripts generados de forma dinámica, durante la evaluación de los primeros.
3. Descargas de documentos HTML.
4. Descargas de objetos externos (ficheros CSS y ficheros con código JavaScript).
5. Ejecución de peticiones AJAX.

En cada celda de la Tabla 3, se muestra primero el valor obtenido en una ejecución normal de la secuencia de navegación, seguido del valor obtenido en la ejecución optimizada.

En la sexta columna, se muestra el tiempo consumido (en la ejecución optimizada) en la búsqueda de los fragmentos irrelevantes, representados mediante las expresiones XPath almacenadas. Este tiempo, se corresponde con el algoritmo detallado en el apartado 3.1.3 para la detección de fragmentos irrelevantes.

El tiempo de detección de fragmentos irrelevantes, forma parte del tiempo de construcción del árbol DOM, y por lo tanto, este tiempo también está incluido dentro del valor de la primera columna (de la ejecución optimizada).

Por último, el valor de la séptima columna muestra el tiempo total de ejecución de la secuencia de navegación.

	BUILDING DOM TREE	EXECUTING SCRIPTS	DOWNLOADING HTML PAGES	DOWNLOADING EXTERNAL OBJECTS	EXECUTING AJAX REQUESTS	CHECKING IRRELEVANT NODES	TOTAL TIME
ALEXA	108/25	1733/952	820/825	1716/947	0/0	3	4426/2782
AMAZON	170/76	1792/873	4700/3544	849/190	940/244	5	8549/5019
APPLESTORE	152/27	2881/1305	193/212	529/391	261/0	4	4228/2003
BARNES&NOBLE	122/42	4171/2173	1910/1912	934/912	0/0	5	7187/5094
BLOOMBERG	118/62	3443/281	889/656	3392/525	0/0	9	7908/1593
CNET	108/75	2575/1063	4710/3949	4097/1839	0/0	42	11563/7065
CNN	100/47	1988/652	3410/1562	3747/468	0/0	2	9294/2779
EBAY	135/106	3102/1442	3716/2580	1318/1005	0/0	47	8377/5274
FLICKR	111/39	1965/530	2999/2569	4221/480	0/0	9	9338/4055
GOOGLENEWS	263/50	2393/584	1345/1387	1757/328	0/0	5	5810/2414
IMDB	193/65	3095/1501	2513/1335	2737/676	756/614	23	9361/4279
LINKEDIN	94/42	1085/499	2634/1752	1686/473	639/0	24	6230/2839
REFERENCE	147/53	3103/1252	3035/1514	6300/1754	0/0	24	12639/4694
REUTERS	179/40	3105/749	2817/620	12485/3983	686/190	17	19341/5621
SOFTONIC	114/52	1614/647	2258/1654	2542/870	0/0	10	6579/3272
SPIEGEL	202/62	1102/567	3432/1101	4791/2519	15/0	15	9570/4297
STACKOVERFLOW	81/41	1471/570	654/758	3255/703	1279/244	6	6770/2341
TARINGA	192/32	3798/980	4054/2244	5083/1241	567/0	8	13746/4546
THEGUARDIAN	126/56	4090/1985	438/397	7486/3023	0/0	15	12219/5604
TRIPADVISOR	217/41	768/53	3225/3207	617/0	0/0	7	4921/3353
W3CSCHOOLS	176/30	2219/0	1927/1108	3787/0	0/0	13	8143/1251
WALMART	171/63	5028/1717	3315/2343	3067/783	910/668	15	12554/5633
WIKIPEDIA	77/50	1977/1261	824/850	4148/2100	0/0	7	7192/4309
WORDPRESS	96/46	568/277	2822/1064	2266/1371	0/0	34	5776/2792
WSJOURNAL	184/231	3894/2402	11041/6984	5800/3877	0/0	72	21028/13621
YAHOO	101/21	926/333	4195/2736	3597/875	0/0	6	8875/4013
YELP	305/36	1538/11	2823/2806	1874/0	0/0	14	6706/2906
TOTAL	**4042/1510 (37,36%)**	**65424/24659 (37,69%)**	**76699/51669 (67,37%)**	**94081/31333 (33,3%)**	**6053/1960 (32,38%)**	**0/441**	**248330/113449 (45,68%)**
TIME SAVINGS	**2532 (1,02%)**	**40765 (16,42%)**	**25030 (10,08%)**	**62748 (25,27%)**	**4093 (1,65%)**	**-441 (-0,18%)**	**134881 (54,32%)**

Tabla 3. Tiempos de ejecución de la secuencia de navegación web.

Cabe destacar que este tiempo no es exactamente igual a la suma de los tiempos de las cinco primeras columnas, dado que faltaría incluir otras pequeñas tareas internas de menor relevancia, pero necesarias para el correcto funcionamiento del motor de renderización (por ejemplo, tiempos de intercambio de información entre motor de renderización y motor de navegación para consultar el estado de la ejecución de la secuencia, etc.).

La Tabla 3 muestra cómo, de media, una ejecución optimizada, consume tan solo el 45.68% del tiempo empleado en una ejecución normal de la misma secuencia de navegación.

Haciendo un desglose por cada una de las tareas involucradas en la carga de las páginas web, y en la ejecución del resto de comandos de la secuencia de navegación, se pueden observar las siguientes mejoras en la ejecución optimizada, con respecto a la ejecución normal:

- La construcción del árbol DOM consume un 37.36% del tiempo.
- La ejecución de los scripts consume un 37.69% del tiempo.
- La descarga de páginas HTML consume un 67.37% del tiempo.
- La descarga de objetos externos consume un 33.3% del tiempo.
- Y la ejecución de secuencias AJAX consume un 32.38% del tiempo.

La última fila, muestra el tiempo total que se ahorra en la ejecución optimizada, y el porcentaje que este tiempo supone, con respecto al tiempo consumido en una ejecución normal. Por lo tanto, esta fila permite repartir por tareas, el ahorro en el tiempo de ejecución.

Como se puede observar, la comprobación realizada sobre cada uno de los nodos para detectar si se trata de un nodo irrelevante, penaliza con tan sólo un 0.18% sobre el tiempo total. Este tiempo es insignificante, comparado con el ahorro total que se alcanza durante la ejecución de la secuencia optimizada.

La explicación de esta diferencia reside en que, la creación de objetos es una operación mucho más costosa que la comparación de cadenas de texto.

El ahorro total del 54.32% en el tiempo de ejecución se divide de la siguiente manera:

- En la tarea de la construcción del árbol DOM, se ahorra un 1.02% del tiempo.
- En la ejecución de scripts se ahorra un 16.42% del tiempo.
- En la descarga de páginas HTML se ahorra un 10.08% del tiempo.
- En la descarga de elementos externos se ahorra un 25.27% del tiempo.
- Y en la ejecución de peticiones AJAX se ahorra un 1.65% del tiempo.

5.1.3 Evaluación de la técnica de ejecución de JavaScript en paralelo

A continuación, se detallan los experimentos realizados para medir la eficiencia de la técnica de evaluación de JavaScript en paralelo.

Para realizar esta evaluación, de nuevo, se ha utilizado un conjunto de sitios web, incluidos en la lista de sitios más visitados según el ranking publicado por Alexa, y se ha ejecutado una secuencia de navegación que carga una página principal, representativa de cada uno de los sitios web.

En primer lugar, se ha realizado una primera ejecución para obtener el grafo de dependencias entre scripts.

A continuación, se ha realizado una ejecución normal de la misma secuencia de navegación, y en último lugar, se ha realizado una ejecución optimizada, que evalúa el código JavaScript en paralelo, utilizando el grafo de dependencias entre scripts calculado al principio.

Para prevenir problemas e inconsistencias, derivados de pequeños cambios en la misma página al acceder a ella en momentos diferentes, cada secuencia de navegación se ha ejecutado 10 veces y los datos que se muestran, representan la media de esas 10 ejecuciones.

Para mejorar la visualización de los resultados, se ha dividido el resultado de este experimento en dos tablas diferentes (Tabla 4 y Tabla 5) y, en cada una de las tablas, se han incluido la mitad de los sitios web, mientras que el resumen se ha añadido al final de la segunda tabla.

La Tabla 4 y la Tabla 5, muestran el tiempo (en milisegundos) que ha empleado el componente de navegación en la ejecución de JavaScript tanto en la ejecución normal, como en la ejecución optimizada.

En la primera columna de ambas tablas, se muestra el tiempo (en milisegundos) de ejecución de JavaScript del componente de navegación evaluando los scripts de forma secuencial.

En la segunda columna, se muestra el tiempo (en milisegundos) de ejecución de JavaScript del componente de navegación, utilizando la técnica de evaluación de JavaScript en paralelo, y entre paréntesis, el porcentaje de mejora con respecto al valor de la primera columna.

	SCRIPTING TIME (MS)	PARALLEL SCRIPTING TIME (MS)
360.CN	392	304 (23%)
ALIBABA.COM	97	81 (17%)
ALLEGRO.PL	425	348 (19%)
AMAZONWS.COM	319	234 (27%)
BBC.COM	203	144 (30%)
BET365.COM	195	116 (41%)
BILD.DE	1098	806 (27%)
BLOGGER.COM	99	65 (35%)
BLOGSPOT.COM	112	66 (42%)
BLOOMBERG.COM	632	529 (17%)
BOOKING.COM	518	438 (16%)
CNET.COM	235	174 (26%)
ENGADGET.COM	447	274 (39%)
FORBES.COM	432	390 (10%)
GITHUB.COM	330	237 (29%)
GIZMODO.COM	80	60 (25%)
GSMARENA.COM	390	335 (15%)
IGN.COM	1228	1036 (16%)
IKEA.COM	81	69 (15%)
IMGUR.COM	686	549 (20%)
INDIATIMES.COM	977	819 (17%)
INSTAGRAM.COM	261	169 (36%)
LEMONDE.FR	305	203 (34%)
LIBERO.IT	396	297 (25%)
LIFEHACKER.COM	86	66 (24%)
LINKEDIN.COM	185	146 (22%)
LIVE.COM	300	266 (12%)
LIVEJOURNAL.COM	106	92 (14%)
MARCA.COM	723	515 (29%)
MASHABLE.COM	145	101 (31%)
MEDIAFIRE.COM	455	357 (22%)

Tabla 4. Tiempo de ejecución de JavaScript (primera parte).

Como se puede observar en estas tablas, la mejora que se produce en el tiempo medio de ejecución de la secuencia de navegación, cuando se utilizan las técnicas de evaluación de scripts en paralelo, es de un 22%, y este porcentaje, varía desde el 6% en el peor de los casos, hasta el 48% en el mejor de los escenarios.

Descartando los resultados que quedan fuera del rango que comprende la media ± la desviación típica, la mejora se sitúa en un 20% y la mediana, se sitúa también en el 20%. Con esta medición se descartan aquellos valores anómalos y que, por lo tanto, pueden considerarse como no representativos o inválidos al ser diferentes, de forma excepcional (como se verá, esta medición se utilizará de aquí en adelante, también en otros experimentos).

En este caso, no se ha incluido en la tabla el tiempo empleado en acceder y manipular el grafo de dependencias entre scripts. Cabe destacar que este tiempo es ínfimo y en la mayoría de las ocasiones no llega a una milésima de segundo

	SCRIPTING TIME (MS)	PARALLEL SCRIPTING TIME (MS)
PETFLOW.COM	727	557 (24%)
PINTEREST.COM	210	173 (18%)
REDIFF.COM	290	245 (16%)
REUTERS.COM	831	725 (13%)
RT.COM	605	478 (21%)
SCRIPBD.COM	689	579 (16%)
SOFTONIC.COM	537	483 (11%)
SOURCEFORGE.NET	504	327 (36%)
SPEEDTEST.NET	770	688 (11%)
STACKEXCHANGE.COM	255	222 (13%)
TAOBAO.COM	191	153 (20%)
TARINGA.NET	1494	1405 (6%)
TECHCRUNCH.COM	249	197 (21%)
THEFREEDICTIONARY.COM	503	424 (16%)
TIME.COM	329	204 (38%)
TRIPADVISOR.COM	131	103 (22%)
TUMBLR.COM	546	418 (24%)
UPLOADED.NET	412	344 (17%)
UPS.COM	1167	1013 (14%)
USATODAY.COM	185	120 (36%)
WARRIORFORUM.COM	346	312 (10%)
WEATHER.COM	516	454 (13%)
WIX.COM	500	433 (14%)
WORDPRESS.COM	63	33 (48%)
WORDREFERENCE.COM	185	120 (36%)
XDA-DEVELOPERS.COM	689	566 (18%)
YAHOO.COM	166	94 (44%)
YOUTUBE.COM	325	298 (9%)
ZIPPYSHARE.COM	563	470 (17%)
AVERAGE		22%
AVERAGE ± STDEV		20%
MEDIAN		20%

Tabla 5. Tiempo de ejecución de JavaScript (segunda parte).

La Tabla 6 y la Tabla 7, muestran un análisis de los diferentes scripts ejecutados en cada una de las secuencias de navegación (para mejorar la visualización, de nuevo se ha dividido el resultado de este experimento en dos tablas diferentes, y en cada una de las tablas, se han incluido la mitad de los sitios web, mientras que el resumen se ha añadido al final de la segunda tabla).

En estas tablas se muestran las siguientes columnas:

1. El número total de scripts que se han evaluado (primera columna).
2. Cuántos scripts han sido ejecutados en los *threads* del pool, que están destinados a la evaluación en paralelo de scripts (segunda columna).
3. Número de scripts detectados como seguros (tercera columna), esto es, sin ninguna dependencia con otros scripts incluidos en la misma página.

De manera adicional, en la segunda y tercera columna, también se muestra entre paréntesis el porcentaje que suponen esos scripts, con respecto al número total de la primera columna.

	TOTAL SCRIPTS	PARALLEL SCRIPTS	SAFE SCRIPTS
360.CN	27	12 (44%)	3 (11%)
ALIBABA.COM	12	7 (58%)	9 (75%)
ALLEGRO.PL	35	15 (42%)	9 (25%)
AMAZONWS.COM	14	9 (64%)	3 (21%)
BBC.COM	101	76 (75%)	15 (14%)
BET365.COM	22	12 (54%)	21 (95%)
BILD.DE	117	35 (29%)	16 (13%)
BLOGGER.COM	13	7 (53%)	6 (46%)
BLOGSPOT.COM	14	6 (42%)	6 (42%)
BLOOMBERG.COM	44	29 (65%)	32 (72%)
BOOKING.COM	36	5 (13%)	9 (25%)
CNET.COM	22	6 (27%)	7 (31%)
ENGADGET.COM	42	25 (59%)	19 (45%)
FORBES.COM	50	21 (42%)	17 (34%)
GITHUB.COM	4	3 (75%)	4 (100%)
GIZMODO.COM	18	8 (44%)	10 (55%)
GSMARENA.COM	77	11 (14%)	22 (28%)
IGN.COM	125	42 (33%)	33 (26%)
IKEA.COM	6	4 (66%)	5 (83%)
IMGUR.COM	41	17 (41%)	17 (41%)
INDIATIMES.COM	76	36 (47%)	13 (17%)
INSTAGRAM.COM	10	4 (40%)	4 (40%)
LEMONDE.FR	54	26 (48%)	4 (7%)
LIBERO.IT	40	16 (40%)	14 (35%)
LIFEHACKER.COM	18	8 (44%)	10 (55%)
LINKEDIN.COM	15	6 (40%)	6 (40%)
LIVE.COM	16	7 (43%)	9 (56%)
LIVEJOURNAL.COM	20	7 (35%)	10 (50%)
MARCA.COM	210	125 (59%)	136 (64%)
MASHABLE.COM	8	4 (50%)	5 (62%)
MEDIAFIRE.COM	28	15 (53%)	7 (25%)

Tabla 6. Scripts ejecutados en paralelo y scripts seguros (primera parte).

Como se puede observar, el número de scripts evaluados en los *threads* del pool varía desde el 14%, en el peor de los escenarios, hasta el 80% en el mejor de ellos.

La media de scripts ejecutados en *threads* del pool es del 45%, la mediana es del 44% y la media, descartando los valores que quedan fuera del rango de la media ± la desviación típica, es también del 45%.

En este experimento, es necesario destacar, que se puede dar el caso de que un script preparado para ejecutarse, se evalúe en el *thread* principal, cuando se da la circunstancia de que este script es el primero de la cola, y el *thread* principal solicita la ejecución de un evento de esta misma cola. Por lo tanto, en algunas fuentes, el número real de scripts que se ejecuta en paralelo, podría ser algo mayor que el valor que se muestra en la segunda columna.

El número de scripts seguros varía desde el 7%, en el peor de los casos, hasta el 100% en el mejor de los escenarios. La media es del 40%, la mediana es del 35% y la media, descartando los valores que quedan fuera del rango de la media ± la desviación típica, es del 37%.

	TOTAL SCRIPTS	PARALLEL SCRIPTS	SAFE SCRIPTS
PETFLOW.COM	32	22 (68%)	7 (21%)
PINTEREST.COM	8	2 (25%)	4 (50%)
REDIFF.COM	27	11 (40%)	12 (44%)
REUTERS.COM	95	15 (15%)	10 (10%)
RT.COM	34	17 (50%)	5 (14%)
SCRIPBD.COM	30	9 (30%)	9 (30%)
SOFTONIC.COM	27	11 (40%)	4 (14%)
SOURCEFORGE.NET	44	22 (50%)	16 (36%)
SPEEDTEST.NET	64	18 (28%)	10 (15%)
STACKEXCHANGE.COM	16	8 (50%)	2 (12%)
TAOBAO.COM	22	12 (54%)	15 (68%)
TARINGA.NET	62	37 (59%)	25 (40%)
TECHCRUNCH.COM	26	21 (80%)	16 (61%)
THEFREEDICTIONARY.COM	86	26 (30%)	26 (30%)
TIME.COM	33	19 (57%)	17 (51%)
TRIPADVISOR.COM	46	26 (56%)	44 (95%)
TUMBLR.COM	36	8 (22%)	11 (30%)
UPLOADED.NET	16	7 (43%)	2 (12%)
UPS.COM	95	73 (76%)	31 (32%)
USATODAY.COM	6	3 (50%)	6 (100%)
WARRIORFORUM.COM	27	8 (29%)	7 (25%)
WEATHER.COM	64	16 (25%)	11 (17%)
WIX.COM	23	7 (30%)	4 (17%)
WORDPRESS.COM	16	10 (62%)	14 (87%)
WORDREFERENCE.COM	8	5 (62%)	4 (50%)
XDA-DEVELOPERS.COM	46	19 (41%)	9 (19%)
YAHOO.COM	42	13 (30%)	17 (40%)
YOUTUBE.COM	23	10 (43%)	9 (39%)
ZIPPYSHARE.COM	27	15 (55%)	8 (29%)
AVERAGE		45%	40%
AVERAGE ± STDEV		45%	37%
MEDIAN		44%	35%

Tabla 7. Scripts ejecutados en paralelo y scripts seguros (segunda parte).

5.1.4 Evaluación de la técnica de estilos CSS bajo demanda

En este apartado, se detallan los experimentos realizados para medir la eficiencia de la técnica de evaluación de estilos CSS bajo demanda.

Por medio de esta técnica, el cálculo de los atributos de visualización, se realiza sólo para algunos de los nodos del árbol DOM, cuando sus propiedades CSS son accedidas durante la evaluación del código JavaScript de la página HTML.

En este primer experimento, se ha ejecutado una secuencia de navegación que realiza la carga de una sola página en varios sitios web (en la mayoría de los casos se ha utilizado la página de inicio). Sobre esta página, se ha calculado el número total de nodos creados y el número de éstos que han requerido del cálculo del estilo CSS.

Para mejorar la visualización de los resultados, se ha dividido el resultado de este experimento en dos tablas diferentes (Tabla 8 y Tabla 9) y, en cada una de las tablas, se han incluido la mitad de los sitios web, mientras que el resumen se ha añadido al final de la segunda tabla.

La Tabla 8 y la Tabla 9, muestran el número total de nodos creados durante el proceso de carga de la página (primera columna), y el número de ellos que han requerido del cálculo de los atributos de visualización, durante el proceso de evaluación de JavaScript (segunda columna). En la tercera columna, se muestra el porcentaje de nodos que ha requerido del cálculo de estilos CSS, con respecto del número total de nodos.

El resultado de este experimento, es que tan solo el 5% del total de nodos creados durante la construcción del documento, ha utilizado la información de visualización, durante la evaluación del código JavaScript.

Calculando la media de los porcentajes (la media de los valores de la tercera columna), sólo el 6% de los nodos requiere del cálculo de la información de visualización y descartando los porcentajes que están fuera del rango de la media ± la desviación típica, el resultado es que sólo el 2% de los nodos requiere del cálculo del estilo CSS.

En este experimento, también se puede observar como hay un número importante de sitios web en los que el uso de los atributos CSS durante la ejecución de JavaScript es del 0% (un total de 19 sitios web, aproximadamente ⅓ del total).

Esto implica que en estos sitios web, el soporte de CSS podría haberse deshabilitado en su totalidad, sin que ello afectase a la correcta ejecución de la secuencia.

Deshabilitando el soporte de CSS en su totalidad, el sistema podría incrementar aún más su rendimiento, al evitar tanto la descarga de las hojas de estilo, como su posterior procesamiento para extraer las reglas que se incluyen en ellas.

	TOTAL NODES	STYLED NODES	STYLED NODES (%)
360.CN	1007	24	2%
ALIBABA.COM	1008	3	0%
ALLEGRO.PL	522	21	4%
AMAZONWS.COM	2098	11	0%
BBC.COM	1192	14	1%
BET365.COM	302	5	1%
BILD.DE	3094	116	3%
BLOGGER.COM	165	13	7%
BLOGSPOT.COM	165	13	7%
BLOOMBERG.COM	2430	15	0%
BOOKING.COM	2310	24	1%
CNET.COM	1542	35	2%
ENGADGET.COM	1410	2	0%
FORBES.COM	1269	3	0%
GITHUB.COM	222	0	0%
GIZMODO.COM	3432	0	0%
GSMARENA.COM	740	29	3%
IGN.COM	1340	1042	77%
IKEA.COM	512	0	0%
IMGUR.COM	854	9	1%
INDIATIMES.COM	1712	80	4%
INSTAGRAM.COM	122	23	18%
LEMONDE.FR	2769	0	0%
LIBERO.IT	1326	43	3%
LIFEHACKER.COM	490	87	17%
LINKEDIN.COM	1066	3	0%
LIVE.COM	166	43	25%
LIVEJOURNAL.COM	404	3	0%
MARCA.COM	12096	202	1%
MASHABLE.COM	586	0	0%
MEDIAFIRE.COM	616	50	8%

Tabla 8. Nodos que requieren el cálculo del estilo CSS (primera parte).

De todos los sitios web incluidos en este primer experimento, sólo uno de ellos presenta un valor excepcionalmente alto (se calcula el estilo CSS del 77% de los nodos del árbol DOM), todos los demás sitios, presentan un porcentaje de nodos, que requieren el cálculo de los estilos CSS, muy bajo.

Por lo tanto, se puede concluir que con esta técnica de optimización se minimiza el uso de memoria y el uso de CPU, dado que un porcentaje muy pequeño del total de nodos del árbol DOM, requiere del uso de la información de visualización, durante la evaluación del código JavaScript.

	TOTAL NODES	STYLED NODES	STYLED NODES (%)
PETFLOW.COM	5147	3	0%
PINTEREST.COM	839	209	24%
REDIFF.COM	1272	347	27%
REUTERS.COM	1436	67	4%
RT.COM	775	46	5%
SCRIPBD.COM	1593	63	3%
SOFTONIC.COM	1793	34	1%
SOURCEFORGE.NET	507	27	5%
SPEEDTEST.NET	751	76	10%
STACKEXCHANGE.COM	910	23	2%
TAOBAO.COM	768	0	0%
TARINGA.NET	1483	167	11%
TECHCRUNCH.COM	1376	1	0%
THEFREEDICTIONARY.COM	1747	208	11%
TIME.COM	951	51	5%
TRIPADVISOR.COM	932	0	0%
TUMBLR.COM	192	11	5%
UPLOADED.NET	132	25	18%
UPS.COM	376	0	0%
USATODAY.COM	1267	3	0%
WARRIORFORUM.COM	1041	150	14%
WEATHER.COM	1054	12	1%
WIX.COM	356	17	4%
WORDPRESS.COM	150	2	1%
WORDREFERENCE.COM	270	21	7%
XDA-DEVELOPERS.COM	1030	35	3%
YAHOO.COM	1259	0	0%
YOUTUBE.COM	1731	18	1%
ZIPPYSHARE.COM	355	80	22%
TOTAL	78460	3609 (6%)	

Tabla 9. Nodos que requieren el cálculo del estilo CSS (segunda parte).

5.1.5 Evaluación del rendimiento de toda la arquitectura del componente de navegación

El último conjunto de experimentos, está orientado a medir la eficiencia de la arquitectura completa del componente de navegación y, para ello, se va a comparar el tiempo de ejecución de la implementación de referencia, con el tiempo de ejecución de otros componentes de navegación.

En primer lugar, se ha seleccionado un componente de navegación basado en un navegador convencional. En este caso, se ha escogido un componente de navegación basado en el navegador Microsoft Internet Explorer.

Este componente, utiliza el API de alto nivel que proporciona Internet Explorer para ejecutar los diferentes comandos que forman parte de la secuencia de navegación web.

En segundo lugar, se ha seleccionado un componente de navegación basado en un navegador a medida y, en este caso, se ha escogido un componente de navegación basado en HtmlUnit, por tratarse de una librería muy popular, de código abierto y utilizada en muchos proyectos reales.

Este componente de navegación basado en HtmlUnit, está desarrollado en Java y para ejecutar las secuencias de navegación, se ha realizado una traducción de los comandos que forman parte de la secuencia, al API particular que ofrece HtmlUnit para manipular e interactuar con el motor de renderización.

La Tabla 10 y la Tabla 11, muestran el resultado del primer experimento (para mejorar la visualización, se ha dividido el resultado de este experimento en dos tablas diferentes y, en cada una de las tablas, se han incluido la mitad de los sitios web, mientras que el resumen se ha añadido al final de la segunda tabla).

En este experimento, de nuevo, se ha seleccionado un conjunto de sitios web reales, todos ellos incluidos en la lista de Alexa de sitios web más visitados.

Sobre cada uno de estos sitios web, se ha generado una secuencia de navegación web que atraviesa varias páginas, o bien siguiendo enlaces, o bien enviando formularios, una vez que éstos han sido rellenados con valores representativos.

Para prevenir inconsistencias derivadas de pequeñas diferencias en una misma página cargada en momentos diferentes, cada secuencia se ha ejecutado 10 veces con cada navegador y el resultado que se muestra, es la media de todas las ejecuciones.

	OPTIMIZED (MS)	NOT OPTIMIZED (MS)	HTMLUNIT (MS)	MSIE (MS)
360.CN	2613	5274 (219%)	6116 (234%)	7974 (305%)
ALIBABA.COM	2759	6249 (226%)	13350 (483%)	12025 (435%)
ALLEGRO.PL	1027	5142 (500%)	9372 (912%)	11185 (1089%)
AMAZONWS.COM	2093	3455 (165%)	9802 (468%)	9722 (464%)
BBC.COM	1220	4194 (343%)	7901 (647%)	6898 (565%)
BET365.COM	1092	1787 (163%)	3195 (292%)	10922 (1000%)
BILD.DE	3450	18003 (521%)	21207 (614%)	15161 (439%)
BLOGGER.COM	1004	4033 (401%)	4409 (439%)	7750 (771%)
BLOOMBERG.COM	2880	5738 (199%)	11149 (387%)	11874 (412%)
BOOKING.COM	5105	6731 (131%)	11378 (222%)	12649 (247%)
CNET.COM	1298	2586 (199%)	3360 (258%)	11586 (892%)
ENGADGET.COM	496	1890 (381%)	5843 (1178%)	9198 (1854%)
FORBES.COM	754	3420 (453%)	3846 (510%)	6706 (889%)
GITHUB.COM	1183	3587 (303%)	3001 (253%)	7254 (613%)
GIZMODO.COM	607	1752 (288%)	2124 (349%)	9109 (1500%)
GSMARENA.COM	1388	8172 (588%)	9084 (654%)	10706 (771%)
IGN.COM	2269	4768 (210%)	4834 (213%)	9135 (402%)
IKEA.COM	199	1105 (555%)	1494 (750%)	5470 (2748%)
IMGUR.COM	2048	14556 (710%)	17402 (849%)	13343 (651%)
INDIATIMES.COM	1517	7732 (509%)	6512 (429%)	7930 (522%)
INSTAGRAM.COM	1367	2419 (176%)	1969 (144%)	7867 (575%)
LEMONDE.FR	405	1950 (481%)	7996 (1974%)	8679 (2142%)
LIBERO.IT	852	2605 (305%)	1930 (226%)	4386 (514%)
LIFEHACKER.COM	1162	1862 (160%)	4298 (369%)	8702 (748%)
LINKEDIN.COM	1507	4405 (292%)	6685 (443%)	5754 (381%)
LIVEJOURNAL.COM	1350	10655 (789%)	19849 (1470%)	17942 (1329%)
MARCA.COM	899	8007 (890%)	10026 (1115%)	9741 (1083%)
MASHABLE.COM	666	1879 (282%)	3089 (463%)	6742 (1012%)
MEDIAFIRE.COM	4271	5935 (125%)	6409 (135%)	7832 (165%)

Tabla 10. Tiempos de ejecución de los diferentes componentes de navegación (primera parte).

En estas tablas se muestran los siguientes elementos:

1. En la primera columna, el tiempo de ejecución (en milisegundos) de la implementación de referencia del componente de navegación web, utilizando las diferentes técnicas de optimización desarrolladas en este trabajo (para ello se ha efectuado de manera previa, una ejecución de la secuencia, con el objetivo de identificar los fragmentos irrelevantes y generar el grafo de dependencias entre los scripts de cada documento).

2. En la segunda columna, se muestra el tiempo de ejecución de la implementación de referencia del componente de navegación web, pero sin utilizar sus capacidades de optimización, esto es, trabajando de forma muy similar a cómo lo hace cualquier otro navegador, y cargando la página web completa y evaluando los scripts de manera secuencial.

3. En la tercera columna, se muestra el tiempo de ejecución del componente de navegación basado en HtmlUnit.

4. En la cuarta columna, se muestra el tiempo de ejecución del componente de navegación basado en el API de alto nivel que ofrece el navegador Microsoft Internet Explorer.

La segunda, tercera y cuarta columna, muestran entre paréntesis el porcentaje que supone ese tiempo de ejecución, con respecto al tiempo empleado por la implementación de referencia, cuando utiliza las capacidades de optimización (primera columna).

	OPTIMIZED (MS)	NOT OPTIMIZED (MS)	HTMLUNIT (MS)	MSIE (MS)
PETFLOW.COM	1283	5433 (423%)	5853 (456%)	6673 (520%)
PINTEREST.COM	5263	6877 (130%)	6463 (122%)	8310 (157%)
REDIFF.COM	1799	5024 (279%)	5865 (326%)	7531 (418%)
REUTERS.COM	7021	18125 (258%)	20031 (285%)	16620 (236%)
RT.COM	2566	7064 (275%)	12033 (468%)	9913 (386%)
SCRIPBD.COM	8005	9673 (120%)	11923 (148%)	14817 (185%)
SOFTONIC.COM	933	3524 (377%)	4821 (516%)	6403 (686%)
SOURCEFORGE.NET	4868	10593 (217%)	12828 (263%)	18260 (375%)
SPEEDTEST.NET	2139	4932 (230%)	6386 (298%)	10823 (505%)
STACKEXCHANGE.COM	2097	5008 (238%)	5541 (264%)	9312 (444%)
TAOBAO.COM	1588	2472 (155%)	8051 (506%)	11610 (731%)
TARINGA.NET	5249	10886 (207%)	8734 (166%)	9695 (184%)
TECHCRUNCH.COM	604	2095 (346%)	5868 (971%)	8551 (1415%)
THEFREEDICTIONARY.COM	915	6307 (689%)	6539 (714%)	7112 (777%)
TIME.COM	4092	6243 (152%)	12103 (295%)	11676 (285%)
TRIPADVISOR.COM	1050	2281 (217%)	6561 (624%)	6997 (666%)
TUMBLR.COM	3469	5054 (145%)	5805 (167%)	7858 (226%)
UPLOADED.NET	1496	3321 (221%)	3682 (246%)	9558 (638%)
UPS.COM	2232	4016 (179%)	3057 (136%)	5846 (261%)
USATODAY.COM	335	1280 (382%)	2161 (645%)	4478 (1336%)
WARRIORFORUM.COM	1540	3091 (200%)	3396 (220%)	7913 (513%)
WEATHER.COM	2045	4457 (217%)	11260 (550%)	10407 (508%)
WIX.COM	1655	3097 (186%)	4024 (241%)	4908 (294%)
WORDPRESS.COM	1975	2770 (140%)	2848 (144%)	10793 (546%)
WORDREFERENCE.COM	832	5086 (611%)	7778 (934%)	6507 (782%)
XDA-DEVELOPERS.COM	3175	5966 (187%)	9793 (308%)	10585 (333%)
YAHOO.COM	3680	5483 (148%)	6590 (179%)	8734 (237%)
YOUTUBE.COM	664	1872 (281%)	2730 (411%)	6334 (953%)
ZIPPYSHARE.COM	1049	2680 (255%)	2228 (212%)	5867 (559%)
AVERAGE		310%	470%	684%
AVERAGE ± STDEV		240%	349%	544%
MEDIAN		246%	378%	534%

Tabla 11. Tiempos de ejecución de los diferentes componentes de navegación (segunda parte).

En las ejecuciones de la implementación de referencia, utilizando las capacidades de optimización, siempre se obtienen los mejores resultados.

Calculando la media de los porcentajes, el tiempo de ejecución de la implementación de referencia, cuando no utiliza las capacidades de optimización, es 3.10 veces más lento (emplea un 310% del tiempo de ejecución de la implementación de referencia, utilizando optimización).

Descartando los resultados que están fuera del rango de la media ± la desviación típica, la implementación de referencia, cuando no utiliza optimización, es 2.4 veces más lenta (emplea un 240% del tiempo total de la versión optimizada) y la mediana indica que es 2.46 veces más lenta.

En cuanto a los resultados obtenidos por los otros componente de navegación, HtmlUnit es el que presenta un rendimiento mejor, aunque de media, es 4.7 veces más lento (emplea un 470% del tiempo) y descartando los resultados que están fuera del rango de la media ± la desviación típica, HtmlUnit es 3.49 veces más lento y la mediana indica que es 3.78 veces más lento.

Por último, el componente de navegación basado en Internet Explorer, es el que obtiene los peores resultados, con una media de 6.84 veces más lento (emplea el 684% del tiempo que tarda la implementación de referencia). Descartando los resultados que están fuera del rango de la media ± la desviación típica, Internet Explorer es 5.44 veces más lento, y la mediana indica que es 5.34 veces más lento.

En este primer experimento, también se puede observar como la implementación de referencia cuando opera como un navegador convencional (sin utilizar sus

capacidades de optimización), obtiene mejores resultados que HtmlUnit en la mayoría de los sitios web.

Del total de 60 sitios web seleccionados para este experimento, HtmlUnit obtiene mejores resultados que la implementación de referencia (sin optimización), tan solo en 8 de ellos.

En el segundo experimento realizado para medir la eficiencia de la arquitectura diseñada en esta tesis doctoral, se han ejecutado pruebas de carga, empleando para ello varias instancias del mismo componente de navegación (por ejemplo, varias instancias de Internet Explorer concurrentes), ejecutándose todas ellas en paralelo durante un período de tiempo constante (el mismo para todos los tipos de navegador).

Este experimento, no se ha realizado sobre los sitios web reales, dado que la gran mayoría, no permiten realizar un número demasiado elevado de peticiones simultáneas.

En lugar de los sitios web reales, se han utilizado versiones locales que emulan el comportamiento de los sitios web originales.

Para conseguir una perfecta simulación de los sitios web reales, se ha seguido el siguiente proceso:

1. En primer lugar, se han descargado las páginas web que atraviesan las secuencias de navegación (también ficheros JavaScript y hojas de estilo externas, etc.).
2. A continuación, todos estos ficheros, han sido modificados para redirigir los enlaces y formularios a las páginas guardadas en local.
3. Y, por último, se han interceptado las peticiones AJAX para que en la versión local del sitio web, devuelvan la misma información, que en las peticiones originales.

Todo este contenido se ha guardado en un servidor web local y a continuación, se han modificado las secuencias de navegación, para que éstas se ejecuten accediendo a este servidor web, en lugar de utilizar el sitio web original.

La primera columna, muestra el número de ejecuciones completadas por componente de navegación cuando utiliza las capacidades de optimización, la segunda columna muestra el número de ejecuciones realizadas por el componente de navegación cuando no utiliza las capacidades de optimización, la tercera columna muestra el número de ejecuciones completadas por el componente de navegación basado en HtmlUnit y, por último, la cuarta columna muestra el número de ejecuciones realizadas por el componente de navegación basado en Microsoft Internet Explorer.

En este caso, un valor más grande en la tabla de resultados, implica un mejor rendimiento. Esto se debe a que las instancias del mismo tipo de navegador,

ejecutándose en paralelo, han sido capaces de completar más ejecuciones de la misma secuencia, en el mismo período de tiempo.

Una vez más, la implementación de referencia del componente de navegación, obtiene los mejores resultados cuando utiliza sus capacidades de optimización.

Comparando con la ejecución normal del componente de navegación, sin utilizar las capacidades de optimización, la implementación de referencia finaliza, en promedio, 4.89 veces más ejecuciones de la misma secuencia de navegación (489%).

Descartando los resultados que superan el rango de la media ± la desviación típica, el número de ejecuciones que finaliza el componente de navegación, utilizando sus capacidades de optimización, es de 3.5 veces más (350%) que cuando no utiliza estas capacidades. Y por último, el valor que indica la mediana es de 3.6 veces más ejecuciones (360%).

Comparando la ejecución de HtmlUnit, la implementación de referencia es capaz de finalizar, de media, 14.25 veces más ejecuciones (1425%).

Descartando los resultados que se salen del rango de la media ± la desviación típica, el número de ejecuciones que finaliza el componente de navegación, es de 9.29 veces más (929%) que HtmlUnit. Y por último, el valor que indica la mediana es de 9.68 veces más ejecuciones (968%).

Comparando la ejecución del componente de navegación que utiliza Microsoft Internet Explorer, la implementación de referencia es capaz de finalizar, de media, 20.57 veces más ejecuciones (2057%).

Descartando los resultados que se salen del rango de la media ± la desviación típica, el número de ejecuciones que finaliza el componente de navegación es de 12.33 veces más (1233%) que Internet Explorer. Y por último, el valor que indica la mediana es de 12.99 veces más ejecuciones (1299%).

En este caso, los resultados del componente de navegación cuando utiliza las capacidades de optimización, son mejores que cuando se accede a los sitios web reales a través de Internet.

Esto es debido a que, al acceder a páginas web almacenadas de manera local, el efecto latencia de la red desaparece, los tiempos de descarga se reducen de forma considerable y el peso del tiempo de uso de CPU es mayor. En este experimento, dado que se han utilizado varios componentes de navegación en paralelo, se ha conseguido que la máquina en la que se han ejecutado las pruebas, llegue al uso máximo de CPU y memoria.

	OPTIMIZED	NOT OPTIMIZED	HTMLUNIT	MSIE
AMAZON.COM	16520	1728 (956%)	552 (2992%)	345 (4788%)
APPLE.COM	6570	3986 (164%)	654 (1004%)	858 (765%)
EBAY.COM	6171	3504 (176%)	1026 (601%)	492 (1254%)
FLICKR.COM	34792	6244 (557%)	909 (3827%)	588 (5917%)
GOOGLE.COM	19719	1737 (1135%)	2413 (817%)	1823 (1081%)
IMDB.COM	6048	2007 (301%)	519 (1165%)	633 (955%)
LINKEDIN.COM	28364	11483 (247%)	4495 (631%)	2110 (1344%)
WALMART.COM	3342	870 (384%)	240 (1392%)	189 (1768%)
WIKIPEDIA.COM	12722	3779 (336%)	1430 (889%)	669 (1901%)
WSJ.COM	4146	651 (636%)	444 (933%)	516 (803%)
AVERAGE		489%	1425%	2057%
AVERAGE ± STDEV		350%	929%	1233%
MEDIAN		360%	968%	1299%

Tabla 12. Pruebas de carga utilizando sitios web locales.

5.2 UTILIZACIÓN EN APLICACIONES INDUSTRIALES

La Plataforma Denodo [DNDPLATF], es una suite de integración de datos distribuidos en tiempo real (también conocida como plataforma de virtualización de datos), comercializada por la empresa Denodo [DENODO].

La Plataforma Denodo, está siendo utilizada en la actualidad por más de 100 compañías en Europa y Estados Unidos, entre las que destacan empresas como: Intel, Time Warner Cable, Eli Lilly, Seagate, Vodafone, Banco Santander o Inditex.

Este producto comercial incluye, entre otras herramientas para la virtualización de datos, un módulo de automatización web, en el que para la ejecución de secuencias de navegación web, es posible utilizar, entre otros componentes, un navegador a medida implementado de acuerdo con la arquitectura propuesta en esta tesis doctoral.

Este módulo de automatización web, se denomina Denodo ITPilot [DNDITP], y permite extraer información de la web, combinando la ejecución de distintos tipos de elementos a través de un flujo de trabajo.

Entre estos elementos ejecutables, se incluyen:

- Secuencias de navegación web.
- Elementos de extracción de datos (que permiten aplicar patrones al código HTML de los documentos obtenidos durante la ejecución de las secuencias de navegación).
- Y otro tipo de elementos de control de flujo, como condiciones, estructuras repetitivas, etc.

La Figura 66, muestra la pantalla de configuración del navegador a medida, una vez que se ha especificado la secuencia de navegación web a ejecutar.

En esta pantalla, es posible optimizar la ejecución de una secuencia de navegación, o bien de forma manual, o bien de forma automática.

Al seleccionar el modo de optimización manual, la herramienta permite modificar algunos de los parámetros del navegador utilizados con más frecuencia (véase el Anexo I, en el que se muestra una descripción detallada de los principales parámetros de configuración manual del componente de navegación).

Al seleccionar el modo de optimización automático, se habilita un botón que permite ejecutar una sola vez la secuencia de navegación web, con el objetivo de recopilar la información de optimización.

Esta información de optimización se almacena de forma persistente, y se utiliza en las siguientes ejecuciones de la misma secuencia, para aplicar las técnicas de optimización detalladas en el capítulo 3: construcción de un árbol DOM minimizado, descartando los fragmentos detectados como irrelevantes, ejecución el paralelo de aquellos scripts que no tienen dependencias entre ellos y ejecución de CSS bajo demanda.

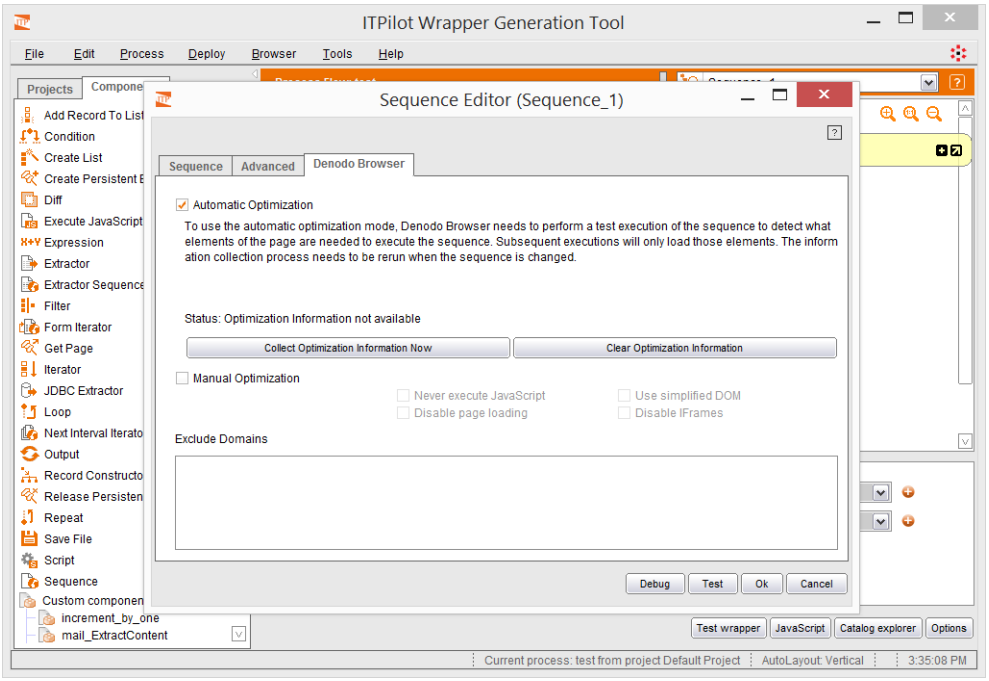


Figura 66. Pantalla de configuración del componente de navegación.

Cuando no se selecciona ninguno de los dos modos de optimización, el navegador a medida ejecuta la secuencia de navegación web, de manera similar a cómo lo hace cualquier otro navegador ad-hoc.

5.3 EVALUACIÓN DEL CUMPLIMIENTO DE OBJETIVOS

En este apartado, se evalúa el grado de cumplimiento de los objetivos de este trabajo, enumerados y definidos todos ellos con anterioridad en el apartado 1.2.

1) Proponer distintas técnicas de optimización, aplicables en la ejecución de secuencias de navegación web.

En este trabajo, se han propuesto tres técnicas de optimización diseñadas para un componente de navegación especializado en la ejecución de tareas de automatización web: construcción de un árbol DOM minimizado, ejecución en paralelo de JavaScript y evaluación de estilos CSS bajo demanda.

La técnica de optimización para construir un árbol DOM minimizado (descrita en el apartado 3.1), permite identificar y descartar, de manera automática, los elementos de una página HTML que no son necesarios para la correcta ejecución de una secuencia de navegación web.

En segundo lugar, la técnica de optimización para la ejecución de JavaScript en paralelo (descrita en el apartado 3.2), permite evaluar scripts de manera simultánea durante la construcción de un documento HTML, cuando estos scripts involucrados no tienen dependencias entre sí.

La técnica de optimización para evaluar estilos CSS bajo demanda (descrita en el apartado 3.3), permite omitir el cálculo de la información de visualización, excepto en aquellos casos en los que esta información es requerida, durante la ejecución del código JavaScript incluido en el documento HTML.

2) Proponer una arquitectura completa de un componente de navegación web, que dé soporte a las técnicas de optimización diseñadas en la fase anterior. Además, se realizará una implementación de referencia de la arquitectura desarrollada.

En este trabajo, se ha diseñado una arquitectura completa para un componente de navegación ad-hoc, con soporte para las diferentes técnicas de optimización diseñadas en la primera fase.

En esta arquitectura, descrita en profundidad en el capítulo 4, se han incluido varios elementos diferenciadores y novedosos, que dan soporte a estas técnicas, como por ejemplo, el componente *optimizador* (que permite gestionar la información con los fragmentos irrelevantes, o el grafo de dependencias entre scripts), o el pool de *threads* que permite ejecutar varios scripts en paralelo.

Esta arquitectura, se ha plasmado en una implementación real, utilizando el lenguaje de programación Java, junto con diferentes librerías de código abierto.

3) Validar las técnicas propuestas, utilizando casos de estudio reales. Cada una de las técnicas de optimización automática que se proponen, debe ser validada por separado midiendo su efectividad, tanto las mejoras obtenidas en el tiempo de ejecución, como en el uso de recursos computacionales (por ejemplo, uso de memoria y uso de la red).

La evaluación experimental de las técnicas de optimización propuestas en este trabajo, se ha detallado en los apartados 5.1.2 (construcción de un árbol DOM minimizado), 5.1.3 (ejecución en paralelo de JavaScript) y 5.1.4 (evaluación de estilos CSS bajo demanda.)

Como se ha podido comprobar en el apartado 5.1.2, los resultados de la evaluación de la técnica de DOM minimizado, han sido de gran calidad.

Esta técnica ha funcionado de manera correcta en todos los sitios web reales utilizados durante la evaluación experimental y además, los recursos consumidos (CPU, memoria y ancho de banda) se minimizan de forma considerable, cuando se descartan los fragmentos irrelevantes.

Por otro lado, el algoritmo de generación de expresiones XPath se ha mostrado muy efectivo, siendo necesarios menos de 2 nodos para identificar a cada uno de los fragmentos irrelevantes (en promedio).

Con eso se ha logrado un objetivo importante, que consiste en que pequeños cambios en el código fuente de la página HTML, no afecten al proceso de localización de los nodos irrelevantes.

La técnica de optimización que permite evaluar JavaScript en paralelo, también ha funcionado de forma correcta en todos los sitios web reales incluidos en los experimentos del apartado 5.1.3.

La mejora en el tiempo de evaluación de JavaScript es de aproximadamente el 20% (con respecto a la ejecución de scripts de manera secuencial, utilizando también la misma implementación de referencia del componente de navegación).

Se puede concluir, por lo tanto, que esta técnica es muy efectiva, sobre todo en los escenarios computacionalmente más complejos (escenarios con mayor uso de CPU, debido al gran volumen de contenidos JavaScript) en los que además, se requiera de un tiempo de respuesta muy reducido.

Los experimentos realizados para validar la técnica de evaluación de estilos CSS bajo demanda, descritos en el apartado 5.1.4, muestran como tan solo un 6% (en promedio) de los nodos totales, requieren del cálculo de los atributos de visualización.

Por lo tanto, esta técnica permite un ahorro en el tiempo de CPU utilizado para el cálculo de esa información, y además, supone también un ahorro en el uso de memoria, al no ser necesario espacio adicional para almacenar objetos con la información de visualización.

El cálculo de la información de visualización en el navegador a medida, se realiza de manera simulada (no existe un proceso de renderización real) y, por lo tanto, algunos de los atributos de visualización, presentan valores diferentes a los obtenidos con los navegadores convencionales.

Esta peculiaridad, se puede observar en el Anexo II de este documento, en el que se muestra el nivel de compatibilidad del componente de navegación, con algunas

de las librerías JavaScript más utilizadas a día de hoy. Este nivel de compatibilidad varía desde el 88% en el peor de los escenarios, hasta el 100% en el mejor de ellos.

Gran parte de estas pruebas de unidad que obtienen un resultado distinto al de los navegadores convencionales, se debe a pequeñas diferencias en el cálculo de los atributos de visualización (por ejemplo, diferencias en el cálculo del tamaño de un nodo del árbol DOM).

No obstante, cabe destacar que estas pequeñas diferencias, en raras ocasiones supone un problema a la hora de ejecutar secuencias de navegación sobre sitios web reales, lo que garantiza el éxito de esta técnica en la práctica totalidad de los escenarios.

4) Validar la arquitectura completa del componente de navegación. En este caso, debe compararse el tiempo de ejecución del navegador web desarrollado a medida, con el tiempo de ejecución de otros componentes de navegación.

Los experimentos para validar la arquitectura completa del componente de navegación se han detallado en el apartado 5.1.5.

En estos experimentos, se ha comparado el tiempo de ejecución de la implementación de referencia de la arquitectura, con respecto al tiempo de ejecución empleado por otros componentes de navegación (HtmlUnit y Microsoft Internet Explorer).

Se han realizado experimentos sobre sitios web reales, y pruebas de carga sobre sitios web simulados en un servidor web local (en estas últimas, se ha alcanzado el uso de memoria y de CPU máximo de la máquina de pruebas).

Estos experimentos, muestran un alto rendimiento en la implementación de referencia cuando utiliza las diferentes técnicas de optimización.

E incluso, si se compara con el rendimiento de los otros componentes de navegación incluidos en los experimentos, la implementación de referencia muestra unos resultados excelentes, cuando no utiliza sus capacidades de optimización.

Para finalizar, una vez que la implementación de referencia ha alcanzado el nivel de madurez suficiente, se ha incluido dentro de un producto comercial.

Por lo tanto, en estos momentos, el componente de navegación está siendo utilizado en proyectos de automatización web reales, muchos de ellos con altos requerimientos de carga de trabajo.

6 DISCUSIÓN, CONCLUSIONES Y LÍNEAS DE TRABAJO FUTURO

Para finalizar esta tesis doctoral, en este capítulo se abordan las conclusiones finales del trabajo realizado.

Para ello, en primer lugar, se realizará una discusión de las técnicas propuestas, comparándolas con otros trabajos previos (apartado 6.1).

A continuación, se repasan las principales contribuciones originales y se presentan las conclusiones de este trabajo (apartado 6.2).

Y para concluir, se esbozan las líneas de trabajo futuro del autor (apartado 6.3).

6.1 DISCUSIÓN

En este apartado, se discuten las aportaciones de este trabajo, con respecto al estado del arte previo, ampliamente detallado en el capítulo 2.

La organización de este apartado está desglosada de la siguiente manera:

En primer lugar, se muestran las similitudes y las diferencias con otras aproximaciones existentes hoy en día, en el ámbito de la automatización web (apartado 6.1.1).

A continuación, se discute la validez de las técnicas de optimización propuestas en este trabajo, y se comparan con otras técnicas desarrolladas en otros sistemas de navegación web ad-hoc (apartado 6.1.2).

Y por último, se analizan los elementos diferenciadores, incluidos en la arquitectura del componente de navegación especializado en la ejecución de secuencias de navegación web (apartado 6.1.3).

6.1.1 Navegadores web convencionales y navegadores web ad-hoc

La mayoría de los sistemas de automatización web existentes hoy en día, utilizan las APIs de alto nivel que proporcionan los navegadores convencionales.

Algunos de estos sistemas son los siguientes: iMacros [IMACROS], Selenium [SLNM], Mozenda [MOZENDA], QEngine [QEGN], Sahi [SAHI], SmartBookmarks [HM07], Wargo [PRAHV02] o PhantomJS [PHAJS].

Esta aproximación, tiene la principal ventaja de que no requiere del desarrollo de un componente de navegación desde cero, y además, garantiza que las páginas a las que se accede de manera automática, se van a comportar de la misma manera que cuando un usuario humano accede a ellas de forma manual.

No obstante, los navegadores convencionales no son aplicaciones desarrolladas pensando en tareas de automatización web, y en ciertos escenarios que requieren de una respuesta en tiempo real, o que requieren de un elevado nivel de concurrencia, con cientos de navegadores ejecutándose al mismo tiempo, esta solución puede no escalar adecuadamente.

A pesar de esto, los navegadores convencionales también utilizan distintos tipos de optimizaciones enfocadas a mejorar el rendimiento.

Por ejemplo, Mozilla Firefox utiliza un procesado especulativo para detectar lo antes posible elementos descargables [MOZSPPAR], y Google Chrome explota el paralelismo utilizando diferentes procesos del sistema operativo para cada una de las ventanas abiertas.

Otros sistemas de automatización web, utilizan una aproximación diferente, que consiste en utilizar navegadores a medida simplificados, diseñados de forma específica para ejecutar tareas de automatización web.

Este es el caso de Jaunt [JAUNT] o Twill [TWILL], sistemas que utilizan clientes HTTP muy sencillos y sin soporte para ejecutar código JavaScript (con lo cual, su funcionalidad es muy limitada y no son soluciones válidas para acceder a los sitios web más dinámicos).

Un último grupo de sistemas, utiliza también navegadores desarrollados a medida, pero en este caso, con amplio soporte JavaScript, hojas de estilo CSS y otras funcionalidades avanzadas, todas ellas disponibles en los navegadores convencionales. Este es el caso de HtmlUnit [HTMUNIT], EnvJS [ENVJS] y ZombieJS [ZOMJS].

Aunque estos sistemas explotan algunas de las peculiaridades de las tareas de automatización web (sobre todo, como no requieren de interfaz gráfica, evitan la renderización del árbol DOM del documento), en todo lo demás, se comportan de manera muy similar a cómo lo hace un navegador convencional, sobre todo en lo que se refiere al procesamiento y carga del documento HTML.

El sistema propuesto en este trabajo, también es un sistema de navegación a medida con soporte para las principales funcionalidades avanzadas, incluidas en los navegadores convencionales (CSS, JavaScript, AJAX, etc.), con la diferencia de que este sistema presenta una serie de técnicas de optimización novedosas y una serie de elementos estructurales diferenciadores, que hasta este momento, no incluye ningún otro componente de navegación.

Las técnicas de optimización propuestas en este trabajo, explotan características específicas de los sistemas de automatización web, no sólo la carencia de interfaz de usuario, sino también otras peculiaridades, como por ejemplo:

- La repetitividad de las secuencias de navegación web (una misma secuencia de navegación, se suele ejecutar en repetidas ocasiones, variando los parámetros de entrada).
- O el hecho de que en la práctica totalidad de los escenarios, para reproducir una secuencia de navegación web, sólo se necesitan ciertos fragmentos de los documentos HTML (existiendo, por lo tanto, una gran cantidad de elementos dentro de los documentos que pueden ser ignorados y descartados).

Como se ha demostrado en los experimentos del capítulo 5, la implementación de referencia del sistema de navegación propuesto, es muy eficiente.

No obstante, el proceso de implementación ha sido costoso, al tener que desarrollar de manera íntegra, un navegador web completo. Aunque una vez que éste ha alcanzado el nivel de madurez suficiente, su rendimiento en tareas de automatización web, es bastante superior al de los otros sistemas con los que ha sido comparado.

6.1.2 Técnicas de optimización específicas de los navegadores web ad-hoc

En este trabajo, se presentan una serie de novedosas técnicas de optimización para un componente de navegación ad-hoc, especializado en la ejecución de secuencias de navegación web.

Ninguno de los sistemas de automatización web existentes hasta el momento, da soporte a ninguna de estas técnicas propuestas.

La técnica de construcción de un árbol DOM minimizado, permite descartar fragmentos del documento HTML, que no son necesarios para ejecutar con éxito la secuencia de navegación. Esta técnica es novedosa y ningún otro sistema de automatización web le da soporte.

La técnica de evaluación de JavaScript en paralelo, tampoco está soportada en ningún otro sistema de automatización web, ni en ninguno de los navegadores convencionales. Todos ellos realizan la evaluación de JavaScript de una manera secuencial.

La mayoría de los sistemas, sólo permiten deshabilitar el soporte de JavaScript de manera manual, mediante alguna opción de configuración.

Esta característica, sólo suele ser de utilidad en un número muy reducido de escenarios, dado que el uso de JavaScript es cada vez mayor en los sitios web reales.

Otros trabajos previos (en este caso, se trata de trabajos en el ámbito de los navegadores convencionales, no en el de la automatización web), explotan otras técnicas de optimización y paralelización, pero ninguna de ellas es capaz de ejecutar JavaScript en paralelo.

Por ejemplo, ZOOM [CSPWM13] es un navegador convencional paralelo, que también realiza un análisis previo del código HTML para detectar elementos descargables y también soporta la aplicación de estilos CSS en paralelo y la compilación de scripts en paralelo (sólo la compilación se ejecuta en paralelo, la ejecución se realiza de manera secuencial).

Otro ejemplo de sistema de navegación convencional, que soporta diferentes técnicas de optimización es Adrenaline [HSSCP12]. En este caso, se trata de un sistema que realiza un procesado en paralelo del documento HTML, tras dividirlo en varios fragmentos y procesar cada uno de estos fragmentos por separado, en un hilo diferente cada fragmento.

En último lugar, la técnica de evaluación de CSS bajo demanda tampoco está soportada en ninguno de los sistemas de automatización web actuales, los cuales, por lo general, sólo permiten habilitar o deshabilitar el soporte de CSS en su integridad, mediante la modificación de alguno de los parámetros de configuración.

6.1.3 Arquitectura del componente de navegación ad-hoc optimizado para la ejecución de secuencias de navegación web

Para dar soporte a las diferentes técnicas de optimización, la arquitectura de navegación diseñada en este trabajo, presenta diferencias importantes con respecto a los sistemas existentes hasta el momento.

En primer lugar, una de las diferencias más relevantes, reside en que el sistema propuesto es un sistema multihilo en lo que se refiere al procesamiento de documentos HTML.

Como se detalla en el capítulo 2, algunos de los principales navegadores convencionales utilizan diferentes *threads* del sistema operativo para gestionar cada una de las ventanas abiertas, pero en lo que se refiere al procesamiento del flujo de datos HTML, éste sigue siendo, en su esencia, un proceso secuencial mono-hilo (aunque algunas de las tareas involucradas, se realizan en hilos auxiliares, por ejemplo, todos los navegadores convencionales realizan las descargas de elementos externos en *threads* auxiliares que se ejecutan en paralelo).

El sistema propuesto, realiza el procesamiento del flujo de datos HTML en diferentes hilos, para implementar la técnica de evaluación de scripts en paralelo.

Por lo tanto, la arquitectura final, ha de dar soporte a esta nueva funcionalidad y para ello, en su diseño se ha incluido un pool de *threads* reutilizables y un *thread* gestor, que controla la asignación de eventos a *threads* libres.

Otra de las diferencias más importantes, reside en que el sistema de navegación debe proporcionar un mecanismo para generar y acceder a la información de optimización (nodos irrelevantes y grafo de dependencias entre scripts).

Para ello, se ha incluido en el diseño de la arquitectura un componente que no tiene equivalencia en ninguno de los sistemas actuales.

Este componente recibe el nombre de *optimizador*, y lo utilizan los otros subsistemas para generar y acceder a los datos de optimización.

Para finalizar, la última de las diferencias estructurales con los sistemas de navegación actuales, reside en el mecanismo utilizado para generar la información de presentación, a partir de las hojas de estilo CSS.

Mientras que en los sistemas actuales, este proceso se realiza de manera incondicional, durante el procesamiento del flujo de datos HTML (tanto en los navegadores convencionales, como en los navegadores ad-hoc más avanzados), en la arquitectura propuesta, la generación de la información de visualización se realiza bajo demanda, durante la evaluación de JavaScript (y sólo para un pequeño número de nodos).

Por ello, el subsistema para el procesamiento de CSS, queda supeditado a su uso desde el motor de ejecución de JavaScript.

6.2 CONCLUSIONES

En este apartado, se presentan las conclusiones de este trabajo.

En primer lugar, en el apartado 6.2.1 se repasan las principales contribuciones originales de esta tesis doctoral y a continuación, en el apartado 6.2.2, se enumeran y justifican las conclusiones finales.

6.2.1 Resumen de las principales contribuciones

Una vez realizada, en el apartado anterior, la discusión con respecto al estado del arte previo, se concluye que este trabajo presenta las siguientes aportaciones principales:

1) *Una técnica de optimización que permite identificar y descartar los fragmentos de los documentos HTML, que no son necesarios para el correcto funcionamiento de la secuencia de navegación web.*

   Mediante esta técnica, se generan árboles DOM muy minimizados, con el consiguiente ahorro, tanto en el tiempo empleado en ejecutar la secuencia de navegación, como en el uso de recursos computacionales (CPU, memoria y ancho de banda).

   Para su desarrollo, es necesaria una ejecución inicial de la secuencia de navegación, en la que se identificarán los nodos irrelevantes.

   Cada uno de estos nodos, se representará mediante una expresión XPath, generada con un algoritmo especialmente diseñado para que el sistema resista pequeños cambios en el código fuente HTML.

   Las siguientes ejecuciones de la misma secuencia, hacen uso de la información obtenida en la ejecución inicial, para construir documentos que tan solo incluyen los fragmentos imprescindibles para ejecutar con éxito la secuencia.

2) *Una técnica de optimización que permite ejecutar en paralelo los scripts contenidos en el documento HTML y que no tienen dependencias entre ellos.*

   Esta técnica, también requiere de una ejecución inicial de la secuencia de navegación. En esta ejecución, se genera un grafo de dependencias entre los nodos de tipo script, tras analizar las interacciones entre los distintos elementos, que forman parte del contexto de ejecución de JavaScript.

   Este grafo permite determinar, para cada uno de los scripts, aquellos otros de los que depende, y que por lo tanto, deben ejecutarse con anterioridad.

   El grafo de dependencias entre scripts, se utiliza en las siguientes ejecuciones de la misma secuencia, para decidir de manera dinámica, qué scripts pueden paralelizar su ejecución.

3) *Una técnica de optimización que permite generar bajo demanda la información de presentación a partir de las hojas de estilo CSS.*

Estos cálculos, se ejecutarán tan solo sobre un conjunto reducido de nodos, en el caso de que se acceda a sus atributos CSS, durante la evaluación del código JavaScript.

Para los demás nodos del árbol DOM, estos cálculos se omiten, con el consiguiente ahorro en el uso de CPU y de memoria.

Esta técnica está basada en la carencia de interfaz gráfica de los navegadores ad-hoc, y al contrario que las otras dos técnicas de optimización, no requiere de ninguna información previa que deba ser calculada con anterioridad en una ejecución inicial de la secuencia de navegación web.

4) *Una arquitectura completa para un componente de navegación especializado en la ejecución de secuencias de navegación web, que da soporte a las técnicas de optimización presentadas en los puntos anteriores.*

   Esta arquitectura multihilo, presenta elementos estructurales novedosos que ningún otro sistema de automatización web incluye hasta el momento y además, ha sido validada con éxito, mediante una implementación de referencia, desarrollada en el lenguaje de programación Java.

   Esta implementación de referencia, ha sido incluida en un producto comercial, por lo que en estos momentos, está siendo utilizada en proyectos de automatización web reales, muchos de ellos con altos requerimientos de carga de trabajo.

6.2.2 Conclusiones obtenidas

A continuación, se enumeran y justifican las principales conclusiones obtenidas tras la elaboración de esta tesis doctoral.

Conclusión 1. El desarrollo de navegadores a medida para tareas de automatización web, es la aproximación más eficiente, aunque también es la más costosa de implementar.

Los navegadores tradicionales, son aplicaciones cliente que no han sido desarrollados pensando en tareas de automatización web, por lo tanto, su rendimiento puede no ser suficiente para dar respuesta a tareas complejas, que requieran de tiempos de ejecución muy reducidos.

Los navegadores a medida, se desarrollan de manera específica para ejecutar tareas de automatización web y pueden evitar ciertas acciones que siempre se ejecutan en los navegadores convencionales durante el proceso de carga de los documentos HTML (por ejemplo, el renderizado del documento).

El principal inconveniente de los navegadores a medida, reside en que su implementación es una tarea costosa, en la que es necesario dar soporte a las funcionalidades disponibles en todos los navegadores tradicionales (CSS, JavaScript, XML, etc.).

Conclusión 2. Los navegadores web a medida, pueden utilizar diferentes peculiaridades de las tareas de automatización web, para desarrollar optimizaciones específicas que no se pueden utilizar en los navegadores convencionales.

Las tareas de automatización web, se basan en la ejecución de secuencias de navegación.

Una misma secuencia de navegación, se ejecuta múltiples veces, por lo que es posible extraer cierta información útil durante una primera ejecución, y utilizar esta información con posterioridad, para optimizar el proceso de construcción de los documentos HTML.

En este trabajo, se utiliza esta peculiaridad para extraer cierta información, que permitirá dar soporte a las técnicas de optimización propuestas.

Para dar soporte a la técnica de construcción de un árbol DOM minimizado, se obtienen los fragmentos del documento que no son necesarios para la correcta ejecución de la secuencia, y para dar soporte a la técnica de ejecución de JavaScript en paralelo, se obtiene un grafo con las dependencias entre los scripts contenidos dentro de un mismo documento HTML.

Por otro lado, los navegadores a medida, son aplicaciones que no están diseñadas para ser utilizadas por personas y por lo tanto, no requieren de interfaz de usuario ni capacidades de renderización.

Esta peculiaridad se utiliza para dar soporte a la técnica de evaluación de CSS bajo demanda.

Conclusión 3. La técnica de optimización propuesta en este trabajo que permite generar un árbol DOM minimizado, descartando los fragmentos irrelevantes, se ha mostrado muy eficiente permitiendo un ahorro en el uso de memoria, en el uso de CPU y en el uso de ancho de banda de la red.

Esta técnica de optimización, permite obtener información sobre las partes de un documento HTML que se pueden eliminar, sin que este hecho afecte al correcto funcionamiento de la secuencia de navegación.

Esta información, se obtiene durante una primera ejecución de la secuencia, se almacena de forma persistente y se utiliza durante las siguientes ejecuciones de la misma secuencia.

Como se ha mostrado en la evaluación experimental, tan solo un 12% de los nodos son necesarios (en promedio), para la correcta ejecución de las secuencias de navegación web. Por lo tanto, el rendimiento del componente de navegación cuando utiliza esta técnica es muy superior, al trabajar con documentos muy minimizados tras descartar los fragmentos irrelevantes.

Conclusión 4. El algoritmo desarrollado para la identificación de nodos del árbol DOM mediante expresiones XPath, se ha mostrado eficiente, además de resistente

a pequeños cambios en el código fuente de los documentos HTML descargados (cuando se accede al mismo documento, en instantes diferentes).

Este algoritmo, se basa en utilizar el mínimo número de fragmentos (nodos) posibles, dentro del camino que va desde el nodo a identificar, hasta el nodo raíz del árbol DOM, pasando por todos sus ancestros.

En los experimentos realizados, han sido necesarios menos de dos fragmentos (en promedio), para identificar a cada uno de los nodos irrelevantes. Por lo tanto, las expresiones generadas serán resistentes a pequeños cambios en los documentos HTML.

Por otro lado, el tiempo empleado en la identificación de estos fragmentos durante la construcción del árbol DOM, es un porcentaje muy pequeño con respecto al tiempo total empleado por el motor de renderización. Por lo tanto, se puede concluir que este algoritmo es muy eficiente.

Conclusión 5. La técnica de optimización propuesta en este trabajo para ejecutar JavaScript en paralelo, permite minimizar el tiempo empleado en la evaluación de scripts. Esta técnica es de gran utilidad, sobre todo en escenarios de elevado uso de JavaScript, que requieren de una respuesta en tiempo real.

Esta técnica de optimización, también necesita de una primera ejecución de la secuencia de navegación, para obtener información sobre las dependencias que existen entre los scripts contenidos en un documento HTML.

Esta información se utiliza en las siguientes ejecuciones de la misma secuencia, para evaluar en paralelo aquellos scripts que no tienen dependencias entre ellos.

Durante los experimentos realizados, el tiempo empleado por la implementación de referencia del componente de navegación cuando evalúa los scripts en paralelo, es un 20% inferior al tiempo empleado por este mismo componente de navegación cuando evalúa los scripts de manera secuencial (operando de manera similar a cómo lo hacen el resto de navegadores).

Conclusión 6. La técnica de optimización propuesta en este trabajo para generar la información de visualización bajo demanda a partir de las hojas de estilo CSS, permite ahorrar tiempo de CPU y uso de memoria y además, para su correcto funcionamiento, esta técnica no requiere del cálculo de ningún tipo de información previa.

Los navegadores a medida, no son aplicaciones cliente con interfaz de usuario, por lo tanto, pueden evitar el cálculo de la información de visualización, salvo cuando esta información es necesaria para la correcta ejecución del código JavaScript.

La técnica de evaluación de estilos CSS bajo demanda desarrollada en este trabajo, permite evitar estos cálculos, salvo cuando éstos son necesarios para la correcta ejecución del código JavaScript.

Durante la evaluación experimental, sólo un 6% de los nodos (en promedio), han requerido del cálculo de la información de visualización, por lo que se puede

concluir que esta técnica permite mejorar tanto el uso de CPU, como el uso de memoria, durante el proceso de construcción del documento HTML.

Conclusión 7: La arquitectura del componente de navegación optimizado para tareas de automatización web propuesta en este trabajo, incluye una serie de componentes novedosos hasta el momento, y su implementación de referencia se ha mostrado mucho más eficiente que los otros sistemas incluidos en la evaluación experimental.

La arquitectura diseñada en este trabajo, muestra algunas similitudes con el diseño de los navegadores tradicionales, pero incluye ciertos componentes novedosos, que permiten dar soporte a las diferentes técnicas de optimización propuestas.

Entre esos elementos innovadores, destacan el componente *optimizador*, que permite manipular los fragmentos irrelevantes y el grafo de dependencias entre scripts, o el pool de *threads*, que permite desarrollar una arquitectura *multi-thread*, para evaluar en paralelo, los scripts que no tienen dependencias entre sí.

Los experimentos realizados, muestran un rendimiento superior cuando la implementación de referencia de la arquitectura utiliza sus capacidades de optimización, tanto si se compara con la misma implementación de referencia sin utilizar sus capacidades de optimización (funcionando de manera similar a cómo lo hacen el resto de navegadores), como si se compara con el rendimiento de otros componentes de navegación incluidos en los experimentos (HtmlUnit y Microsoft Internet Explorer).

6.3 LÍNEAS DE TRABAJO FUTURO

En este apartado, se describen las principales líneas de trabajo actuales y futuras del autor.

6.3.1 Generación de manera dinámica de la información de optimización

Las técnicas de optimización para la construcción de un árbol DOM minimizado y para la ejecución de JavaScript en paralelo, requieren de una ejecución inicial de la secuencia de navegación, en la que se genera la información necesaria para aplicar dichas técnicas.

Si las páginas web a las que se accede sufren modificaciones, es posible que algunos de los fragmentos irrelevantes (identificados en la ejecución inicial de la secuencia y almacenados de manera persistente), se vean modificados o reemplazados por otros diferentes, dentro del mismo documento HTML.

Esto puede ocasionar que la información de optimización pierda su validez y que, con el paso del tiempo, ninguno de los fragmentos irrelevantes almacenados de manera persistente, esté presente dentro del documento HTML, con lo que los algoritmos de optimización pueden perder su efectividad, de manera paulatina.

En el caso de la técnica de evaluación de JavaScript en paralelo, cuando el sistema no es capaz de localizar alguno de los scripts contenidos en el grafo de dependencias, o cuando aparecen nuevos scripts dentro del documento, este grafo se invalida de forma automática y no se vuelve a utilizar.

Cuando se da esta situación, es necesario ejecutar de nuevo la secuencia de manera manual, para regenerar la información de optimización y sobrescribir los datos guardados.

Si el sistema fuese capaz de regenerar de manera automática la información de optimización, supondría una mejora importante en el componente de navegación.

Este proceso de regeneración automática, se podría disparar cuando el algoritmo de identificación de nodos irrelevantes detecte un alto porcentaje de fragmentos no localizados en el código fuente HTML, o cuando el sistema detecte variaciones en los nodos de tipo script.

6.3.2 Mejoras en los algoritmos de identificación de nodos y de documentos

Como se detalla en capítulos previos, para identificar de manera única a los nodos del árbol DOM, se utilizan expresiones XPath que en la mayoría de las ocasiones, hacen uso de los atributos de estos nodos.

En algunos escenarios, el algoritmo de selección de atributos pierde su efectividad, al no ser capaz de detectar aquellos atributos que varían de manera dinámica, cada una de las veces que se carga el mismo documento.

Un ejemplo de esta situación, es cuando el identificador de los nodos del árbol DOM se genera de manera dinámica, durante la ejecución del código JavaScript y su valor

depende de algún identificador de sesión (este identificador será diferente en cada una de las ejecuciones).

Una mejora en el sistema, consistiría en que el componente de navegación fuese capaz de detectar estos escenarios y descartar del proceso de identificación de nodos, aquellos atributos que no tienen un valor constante.

Algo similar sucede con el proceso de identificación de documentos.

Como se detalla en el apartado 3.1.4, para identificar un documento de manera única, se utiliza su URL de acceso junto con los nombres de los parámetros de la petición HTTP GET o POST.

Este mecanismo de identificación tampoco resulta efectivo, cuando la URL contiene fragmentos que varían cada vez que se accede al documento.

Un ejemplo de este escenario es cuando el identificador de sesión, se incluye en la URL de un documento.

El proceso de identificación de documentos, es capaz de detectar algunos de los principales mecanismos estándar de generación de identificadores de sesión, pero existen servidores web que no siguen ningún estándar y generan los identificadores de sesión de una forma no estándar.

Una mejora en el componente de navegación, podría consistir en detectar estos escenarios no estándar y generar patrones que permitan identificar las partes variables en las URL de los documentos.

6.3.3 Desarrollo de nuevas técnicas de optimización

Por último, como parte del trabajo actual y futuro del autor, está el desarrollo de nuevas técnicas de optimización, también orientadas en mejorar el rendimiento del componente de navegación.

De manera similar a las técnicas expuestas en este trabajo, esta investigación sobre nuevos algoritmos de optimización está enfocada en dos vertientes diferentes: por un lado, intentar realizar el mínimo número de acciones, sin que ello afecte a la ejecución de las secuencias de navegación, y por otro lado, intentar ejecutar en paralelo el máximo número de operaciones posibles.

Como parte de la investigación actual del autor, se encuentra también la combinación de las técnicas ya desarrolladas. Por ejemplo, un desarrollo en curso consiste en detectar en la ejecución inicial de la secuencia, qué nodos necesitan el cálculo del estilo CSS y almacenar dicha información utilizando expresiones XPath. De esta forma, en las siguientes ejecuciones de la secuencia, se puede realizar el cálculo del estilo CSS en paralelo, durante el proceso de construcción del árbol DOM (antes de la fase de ejecución de JavaScript).

7 ANEXO I: PRINCIPALES PARÁMETROS DE CONFIGURACIÓN DEL COMPONENTE DE NAVEGACIÓN

En este anexo, se detallan los principales parámetros de configuración que se pueden establecer en el componente de navegación para modificar su comportamiento por defecto.

En un primer momento, los valores por defecto de estos parámetros se establecen en la capa de persistencia de datos (implementada mediante un fichero con propiedades, base de datos, configuración JNDI, etc.).

La práctica totalidad de los parámetros se pueden modificar de manera dinámica, en cualquier punto de la ejecución de la secuencia, a través de una URL especial de configuración.

```
Navigate(about:config?CHARSET=UTF-8&HTTP_VERSION=1.1);
```

Figura 67. URL de configuración dinámica del componente de navegación.

Esta URL especial de configuración es *about:config,* y se puede inyectar en cualquier punto de ejecución de la secuencia a través de un comando de navegación.

La Figura 67 ilustra un ejemplo de su funcionamiento.

En este ejemplo, se modifican de manera dinámica dos parámetros de configuración (parámetro *CHARSET* con valor *UTF-8* y parámetro *HTTP_VERSION* con valor *1.1*).

Utilizando esta URL especial, tan sólo se modifica la configuración del navegador que ejecuta el comando, y de esta manera, es posible establecer diferentes parámetros para cada uno de los navegadores (esta configuración dinámica, se realiza en memoria y no se hace persistente).

7.1 PARÁMETROS DE CONFIGURACIÓN GENERALES

Este primer bloque de parámetros de configuración, incluye parámetros generales para habilitar o deshabilitar funcionalidades específicas de alto nivel.

1. ***JAVASCRIPT_ENABLED***: permite activar o desactivar la ejecución de JavaScript en el componente de navegación.

   Cuando se desactiva la evaluación de JavaScript utilizando este parámetro, se descarta la ejecución de cualquier tipo de contenido JavaScript, y también se evita la descarga de cualquier objeto externo que haga referencia a contenido JavaScript.

2. ***SCRIPT_EVALUATION_ENABLED***: permite activar o desactivar la evaluación de los scripts contenidos dentro de un documento HTML.

   Este parámetro, no deshabilita otro tipo de contenido JavaScript, como por ejemplo, aquel que se puede generar desde alguno de los comandos que forman parte de la secuencia de navegación web.

3. ***SCRIPT_DOWNLOAD_ENABLED***: permite activar o desactivar la descarga de documentos JavaScript externos.

   En caso de que estas descargas se desactiven, sólo se evaluarán los nodos de tipo script que incluyan el código dentro del propio documento HTML.

4. ***CACHE_ENABLED***: permite activar o desactivar la caché de documentos externos (sobre todo ficheros JavaScript y hojas de estilo CSS).

   Este parámetro, podría ser útil para desactivar la caché de los navegadores que acceden a un sitio web en particular, en el que no se desea guardar de forma persistente ningún tipo de información.

5. ***SCRIPT_CACHE_ENABLED***: permite activar o desactivar la caché de scripts compilados.

   Para desarrollar esta funcionalidad, se utilizan las capacidades de compilación de Mozilla Rhino. Esta librería permite compilar a *bytecode*, una cadena de texto con contenido JavaScript.

   Esta caché contendrá cualquier tipo de script que se ejecute en el navegador:

   - Scripts externos descargables (scripts con atributo *src*).
   - Scripts contenidos dentro de la página HTML.
   - Scripts generados de manera dinámica, durante la evaluación de otros scripts.

6. ***JAVA_ENABLED***: permite activar o desactivar la ejecución de Applets Java.

7. ***ACTIVEX_ENABLED***: permite activar o desactivar la creación de objetos ActiveX.

El componente de navegación, soporta algunos de los ActiveX predefinidos en el navegador Microsoft Internet Explorer, por ejemplo, MSXML2.XMLHTTP, MSXML2.DOMDOCUMENT, MSXML2.XSLTEMPLATE, etc.

8. ***DOCUMENT_LOAD_ENABLED***: permite activar o desactivar la generación del árbol DOM a partir del código fuente de la página web.

   Esta opción de configuración, permite que el componente de navegación funcione como un cliente HTTP muy sencillo, pero extremadamente rápido cuando se dan las circunstancias adecuadas:

   - La secuencia de navegación no requiere de la ejecución de comandos de manipulación del árbol DOM (por ejemplo, cuando sólo tiene comandos de navegación y no de emisión de eventos).
   - Y por otro lado, la información requerida está contenida en el código fuente HTML, y no es necesario ejecutar el JavaScript de la página para generar esa información de manera dinámica.

9. ***BUILD_MINIMIZED_DOM***: permite generar un árbol DOM minimizado, utilizando un conjunto predefinido de nodos de los siguientes tipos: enlaces, formularios, entradas de formularios (nodos de tipo *INPUT*, *SELECT*, *OPTION* y *TEXTAREA*), y scripts.

   Todos los demás nodos, con etiquetas distintas a las mencionadas con anterioridad, se descartan de manera automática.

   Esta opción de configuración no está relacionada con la técnica de optimización de construcción de un árbol DOM minimizado descrita en este trabajo, aunque su naturaleza es similar, con la diferencia de que los nodos a incluir en el árbol DOM, se encuentran en una lista predefinida y no se obtienen de manera dinámica, tras un análisis previo de la secuencia de navegación.

   El número de sitios web en los que se puede activar este parámetro es limitado, aunque en aquellos en los que es posible hacerlo, el rendimiento del componente de navegación se incrementa de manera exponencial.

10. ***IFRAMES_ENABLED***: permite activar o desactivar la carga de marcos (nodos de tipo *FRAME* o *IFRAME*).

    Los marcos funcionan de forma similar a las ventanas y cada uno de ellos contiene un documento en su interior.

    Cuando se desactiva la carga de estos nodos, la descarga asociada al marco se descarta y, en su lugar, se inserta un documento predefinido con un número muy reducido de nodos (nodo *HEAD*, *TITLE* y *BODY* vacío).

11. ***MAX_FRAMES***: permite establecer el número máximo de marcos a cargar dentro de un mismo documento.

Cuando se ha alcanzado el número máximo, los nuevos marcos contendrán tan solo un documento predefinido, que contiene un número muy reducido de nodos.

12. ***MAX_FRAME_DEPTH***: permite establecer la profundidad máxima (nivel de anidamiento), durante la construcción del árbol DOM de la página.

    Cuando este nivel se ha alcanzado, cada marco adicional contendrá sólo un documento predefinido.

13. ***MAX_WINDOWS***: permite establecer el número máximo de ventanas que puede crear un navegador.

    Cuando este número máximo se ha alcanzado y se solicita la creación de una nueva ventana, se lanza una excepción para indicar que no es posible crear más objetos de este tipo.

14. ***ALLOW_OPEN_NEW_WINDOWS***: permite activar o desactivar el bloqueador de ventanas emergentes.

    De manera similar a cómo funciona el bloqueador de ventanas emergentes incluido en los navegadores convencionales, cuando está activado, cada nuevo intento de crear una ventana (por ejemplo, utilizando la función predefinida *open*, del objeto *window*), producirá una excepción de JavaScript.

15. ***META_TAGS_ENABLED***: permite activar o desactivar el soporte para los nodos de tipo META.

    Un ejemplo en el que este parámetro podría ser de gran utilidad, es en aquellos sitios web que refrescan de forma automática la página que tiene cargada el navegador, utilizando un nodo de tipo META con un atributo *http-equiv* con valor *refresh* y con un tiempo de refresco definido en el atributo *content*.

    Desactivando el soporte de etiquetas META, también se pueden prevenir estas navegaciones de refresco en caso de que no sean deseadas.

16. ***SCRIPT_URL_SIMPLIFIED***: permite simplificar la URL de un objeto JavaScript externo eliminando de ella los parámetros.

    Esta opción de configuración, puede ser de gran utilidad en aquellos sitios web que añaden parámetros en la URL de todos los elementos descargables, tan solo para prevenir que el navegador utilice la caché con estos elementos.

17. ***CHARSET***: permite especificar de manera manual el juego de caracteres que va a utilizar el navegador, para procesar los distintos ficheros que se pueden descargar de un sitio web.

    El orden que utiliza el navegador para seleccionar un juego de caracteres u otro, es el siguiente:

- Primero utiliza el juego definido en este parámetro.
- En caso de que no esté definido, intenta autodetectarlo (en primer lugar, intenta obtener el juego definido en el contenido HTML y en segundo lugar, intenta obtener el juego definido en las cabeceras HTTP).
- Y en caso de que no sea capaz de autodetectarlo, utiliza el juego por defecto del protocolo HTTP.

18. ***CHARSET_AUTODETECT_ENABLED***: para mejorar la eficiencia, este parámetro permite desactivar la autodetección del juego de caracteres establecido en el contenido HTML (por ejemplo, establecido en alguna de las etiquetas META).
19. ***MIMETYPES***: permite establecer los tipos MIME soportados en el navegador.

    El valor por defecto de este parámetro es:

    - *text*: documentos de texto que se cargan en el componente de navegación de manera directa, sin más procesamiento.
    - *image/ audio/ video/:* ficheros de imágenes, audio y vídeo en formato binario.

      El componente de navegación, carga estos ficheros pero no realiza ningún procesamiento sobre ellos.
    - *application/word application/x-msword application/msword*: ficheros de Microsoft Word.

      Una vez que el componente de navegación tiene cargado un documento en este formato, se pueden ejecutar diversos comandos de conversión a formato HTML y generar el árbol DOM.

      Para realizar la conversión de documentos de Microsoft Word a formato HTML, el componente utiliza librerías de conversión de Open Office.
    - *application/pdf application/x-pdf*: ficheros en formato PDF.

      Una vez que el componente ha cargado estos ficheros, es posible ejecutar comandos de conversión para transformar el contenido PDF a HTML y generar el árbol DOM.

      Para realizar la conversión de documentos en formato PDF a texto HTML, el componente utiliza las librerías de conversión de Apache PDFBox.
    - *application/vnd.ms-excel*: ficheros en formato Microsoft Excel.

Una vez que el componente ha cargado el documento Excel, también es posible convertirlo a HTML y generar el árbol DOM mediante la utilización de diversos comandos específicos que se pueden incluir dentro de la secuencia de navegación.

Para realizar la conversión de documentos de Microsoft Excel a formato HTML, el componente utiliza librerías de conversión de Open Office.

- *application/xml*: documentos de tipo XML. Estos documentos se procesan y se genera su árbol DOM.

Cualquier otro tipo de contenido enviado por un servidor web, será procesado emulando una ventana de descarga (simulando la ventana de descarga de los navegadores tradicionales), y en caso de que los parámetros *DOWNLOAD_DIRECTORY* y *DOWNLOAD_FILE* estén definidos, el fichero descargado de la red se guardará en esa ubicación.

20. ***DOWNLOAD_DIRECTORY***: permite establecer el directorio por defecto, para descargar los tipos de datos (*MIMETYPES*) que no están soportados en el navegador.

21. ***DOWNLOAD_FILE***: permite establecer el nombre del fichero por defecto, para descargar los tipos de datos (*MIMETYPES*) que no están soportados en el navegador.

22. ***COMMENTS_ENABLED***: permite descartar los nodos de tipo comentario del árbol DOM.

    Eliminando los comentarios, se puede crear un árbol DOM más pequeño y también permite desactivar algunas funcionalidades propietarias de Microsoft Internet Explorer, como por ejemplo los comentarios condicionales.

    Por medio de esta funcionalidad, Microsoft Internet Explorer permite definir sentencias (con una sintaxis propietaria), dentro de nodos de tipo comentario incluidos en el documento HTML.

23. ***PROFILING_ENABLED***: con este parámetro se puede activar el analizador de rendimiento del navegador.

    Este módulo, es el encargado de recopilar información sobre las operaciones que realiza el componente de navegación, durante la ejecución de la secuencia de navegación web (por ejemplo, toda la información que se muestra en las tablas de los experimentos del capítulo 5). Entre la información que es posible capturar, destacan los siguientes elementos:

    - Número de descargas de los distintos tipos de documentos (documentos HTML, documentos CSS, documentos JavaScript, peticiones AJAX, etc.).

- Tiempo consumido por cada uno de los subsistemas auxiliares (procesado de HTML, ejecución de scripts, evaluación de CSS, descargas de la red, etc.).
- Cantidad de recursos consumidos (número de nodos creados, número de scripts evaluados, etc.).

24. ***MAX_HISTORY_SIZE***: este parámetro permite definir el número máximo de documentos almacenados dentro del objeto predefinido *history* (historial).

    Cada ventana, contendrá un objeto historial, en el que se guardan los últimos documentos cargados dentro de esa misma ventana.

    Estableciendo un valor de configuración pequeño para este parámetro, se minimiza el uso de memoria. Incluso podría establecerse un valor 0, para desactivar completamente el histórico.

25. ***CSS_ENABLED***: este parámetro permite activar o desactivar el soporte para hojas de estilo CSS. Desactivar el soporte CSS, implica los siguientes cambios en el comportamiento del motor de renderización:
    - Los documentos externos que hagan referencia a hojas de estilo no se descargarán.
    - Los nodos de tipo *STYLE*, incluidos dentro del propio documento HTML, no se procesarán.
    - Cuando desde el motor de JavaScript se consulta el valor de alguno de los atributos de visualización, el subsistema CSS devolverá valores predefinidos (valores fijos, y no calculados de manera dinámica).

26. ***CSS_DOWNLOAD_ENABLED***: permite activar o desactivar la descarga de hojas de estilo CSS. Cuando la descarga de hojas de estilo está desactivada, sólo se evaluarán los nodos de tipo *STYLE*, contenidos dentro del documento HTML que se está procesando.

    Del mismo modo, cuando desde el motor de JavaScript se consulte el valor de alguno de los atributos de presentación, los valores se calcularán de manera dinámica utilizando sólo las reglas disponibles.

27. ***HTML_NORMALIZATION_ENABLED***: permite activar o desactivar la construcción de un documento HTML válido, realizando las modificaciones que sean necesarias para que el árbol DOM generado sea lo más correcto posible.

    Por ejemplo, si el documento HTML contiene dos etiquetas de tipo BODY, el árbol DOM se generará con sólo un nodo BODY, y en su interior, se añadirá el contenido de los dos nodos originales.

28. ***HTML_TAG_BALANCER_ENABLED***: permite activar o desactivar el balanceador de etiquetas.

Este balanceador, actuará a la hora de procesar el documento HTML, cuando alguna de las etiquetas no se encuentre cerrada de manera correcta, dentro del código fuente original.

El balanceador de etiquetas, será el encargado de tomar la decisión de qué hacer con las etiquetas mal cerradas.

29. ***MSIE_VERSION***: permite configurar la versión de Microsoft Internet Explorer que simula el componente de navegación.

    Esto afectará al valor por defecto de otros parámetros de configuración (por ejemplo, al *USER_AGENT*), y afectará también al comportamiento del componente, a la hora de evaluar el contenido JavaScript.

30. ***AJAX_ENABLED***: permite activar o desactivar la ejecución de peticiones AJAX emitidas durante la evaluación del código JavaScript:

    - Utilizando el objeto predefinido *XmlHttpRequest.*
    - Utilizando alguno de los objetos ActiveX soportados en Microsoft Internet Explorer.

    En caso de deshabilitar las peticiones AJAX, el contenido que devolverá la petición HTTP se podrá configurar con otro parámetro de configuración, y también se podrá configurar el código HTTP de la respuesta (por defecto será 200, con contenido vacío).

31. ***WEB_DIALOGS_ENABLED***: permite habilitar o deshabilitar la creación de diálogos, tanto modales (*showModalDialog*) como no modales (*showModelessDialog*).

    El componente también permite tratar estos diálogos como si fuesen ventanas normales.

7.2 PARÁMETROS DE CONFIGURACIÓN PARA MODIFICAR LAS PETICIONES HTTP

A continuación, se detallan los parámetros de configuración relacionados con las peticiones HTTP que emite el componente de navegación, durante la ejecución de la secuencia.

1. ***HTTP_VERSION***: permite configurar la versión del protocolo HTTP, sobre la que operará el componente de navegación.

2. ***MAX_DOWNLOAD_TIME***: tiempo máximo permitido para la finalización de una descarga de un documento de la red.

   Una vez transcurrido este tiempo, el componente de navegación permite establecer un manejador de errores para gestionar el *timeout* en la descarga (existen manejadores que permiten ignorar el error en la descarga, manejadores que lanzan una excepción dependiendo del tipo de documento que se está descargando, etc.).

3. ***PROXY_LOGIN***: permite establecer el usuario para la autenticación en un proxy.

   El componente de navegación soporta autenticación básica (con opción para autorización preventiva, y minimizar así el número de peticiones HTTP), *Digest* y *NTML*.

4. ***PROXY_PASSWORD***: permite establecer la contraseña para la autenticación en un proxy.

5. ***PROXY_DOMAIN***: permite establecer el dominio de autenticación en un proxy (el dominio podría utilizarse, por ejemplo, en la autenticación de tipo NTLM).

6. ***PROXY_AUTOCONFIGURATION***: ruta con la URL del script de autoconfiguración del proxy (soporte para PAC o *proxy autoconfiguration*). Esta ruta puede ser un fichero local, una dirección HTTP o un fichero localizado en un servidor FTP.

7. ***USER_AGENT***: este parámetro permite modificar la cabecera *User-Agent* en las peticiones HTTP que emite el componente de navegación.

   Este parámetro es muy útil en sitios web que ofrecen una versión diferente de las páginas HTML, dependiendo del dispositivo que se conecte. Estos contenidos, suelen incluir la misma información que la versión de escritorio, pero suelen ser más ligeros y con menos carga de JavaScript en las versiones para dispositivos móviles.

8. ***BANNED_SITES***: permite establecer una serie de sitios web a los cuales no se realizarán peticiones HTTP de ningún tipo.

   Este parámetro es útil para descartar las peticiones de redes sociales (por ejemplo, para el rastreo de actividad), anuncios de publicidad, contadores de visitas, etc.

9. ***ALLOWED_SITES***: permite establecer un conjunto de sitios web, a los que se permite realizar peticiones HTTP.

   Cuando este parámetro está definido, toda petición a un sitio web diferente de los que se incluyen en esta lista, será descartada.

10. ***TARGET_SITE_ONLY***: cuando este parámetro está activado, el componente de navegación descartará todas las peticiones HTTP que no se envíen al sitio web al que se ha realizado la primera navegación (este sitio web se autodetecta, a partir del primer comando de navegación incluido dentro de la secuencia de navegación web).

11. ***EXTRA_HEADERS***: este parámetro permite especificar una lista de cabeceras adicionales que se incluirán en las peticiones HTTP que emite el componente de navegación.

12. ***ENABLE_COMPRESSION***: habilita la compresión gzip de los contenidos descargados de la red y, para ello, se establecen de manera automática las cabeceras HTTP apropiadas. Este parámetro está habilitado por defecto.

13. ***COOKIES_ENABLED***: este parámetro habilita el soporte de cookies dentro del componente de navegación.

14. ***INVALID_COOKIES_ENABLED***: permite establecer una política ante aquellas cookies (tanto las que envía el servidor web en las cabeceras HTTP, como las que se establecen durante la ejecución de JavaScript) que tienen algún parámetro incorrecto.

    Por defecto, estas cookies se descartan aunque puede modificarse el comportamiento por defecto, para ignorar los campos inválidos y aceptar la cookie.

15. ***IGNORE_DOWNLOAD_ERRORS***: permite establecer la política a seguir, ante los fallos en las descargas de documentos de la red.

    Por defecto, ante cualquier fallo en alguna descarga, la ejecución de la secuencia de navegación se detiene y se considera fallida. Este comportamiento se puede modificar para ignorar errores HTTP (códigos de error 400 o 500).

16. ***CLIENT_KEYSTORE***: permite establecer la ruta del almacén de claves que se utilizará para la autenticación del componente de navegación, utilizando un certificado de cliente.

17. ***CLIENT_KEYSTORE_TYPE***: permite especificar el tipo del almacén de claves especificado en el parámetro anterior (por ejemplo, JKS o PKCS12).

18. ***CLIENT_KETSTORE_PASSWORD***: permite especificar la contraseña del almacén de claves, especificado mediante el parámetro anterior.

19. ***CLIENT_CERTIFICATE_ALIAS***: permite especificar el nombre del certificado que se incluirá en la autenticación mediante certificado en el cliente.

Este parámetro, es necesario cuando el almacén de claves incluye varios certificados que se podrían utilizar en la autenticación cliente.

20. ***IE_DISK_COOKIES_DIRECTORY***: permite establecer el directorio en el que el navegador Microsoft Internet Explorer almacena las cookies persistentes. En el caso de que este parámetro esté definido, el componente de navegación lee esas cookies con un conector especial y las utiliza como propias.

7.3 PARÁMETROS DE CONFIGURACIÓN DEL MOTOR DE JAVASCRIPT

En este apartado, se detallan los parámetros relacionados con la configuración de bajo nivel, del motor de ejecución de JavaScript.

1. ***IGNORE_JAVASCRIPT_ERRORS***: este parámetro permite establecer la política a seguir ante los errores que se pueden producir durante la evaluación del código JavaScript.

   La mayor parte de estos errores, no suponen un problema para el correcto funcionamiento de la secuencia de navegación web, por lo que podrían ignorarse y continuar ejecutando los demás scripts.

2. ***IGNORE_JAVASCRIPT_COMPILATION_ERRORS***: permite establecer la política a seguir ante los errores de *parsing* de un documento JavaScript.

   En algunos casos sencillos, incluso es posible arreglar el error y ejecutar el script.

3. ***JS_OPTIMIZATION_LEVEL***: este parámetro permite establecer el nivel de optimización, cuando se genera código binario (*bytecode* Java), a partir de una cadena de texto que contiene código JavaScript.

   El nivel de optimización puede ir desde 0 hasta 9. Un nivel mayor de optimización, también implica un mayor tiempo de computación.

   Por lo tanto, debe tenerse en cuenta si los scripts que se están compilando se van a reutilizar. Esto significa que, si la caché de scripts compilados está deshabilitada, un nivel demasiado alto, incluso puede llegar a ser contraproducente.

   Estableciendo un valor negativo, se deshabilita la compilación, y los scripts se interpretan de manera directa.

4. ***APP_NAME***: permite configurar el nombre de la aplicación que usará el objeto predefinido *navigator* (propiedad *appCode*).

   El valor que utilizan los navegadores modernos (Firefox, Chrome, Safari y Microsoft Internet Explorer 11) es "*Netscape*". Las versiones anteriores de Microsoft Internet Explorer utilizan el valor "*Microsoft Internet Explorer*".

5. ***APP_CODE***: de manera análoga, este parámetro permite especificar el valor de la propiedad *appCodeName* que utilizará el objeto predefinido *navigator*.

   Los navegadores convencionales, utilizan el valor "*Mozilla*".

6. ***USER_LANG***: permite establecer el idioma que utilizará el navegador.

   Este parámetro afectará, entre otros, a la propiedad *userLanguage* del objeto predefinido *navigator*.

Cuando este parámetro no tiene un valor definido, se obtiene el lenguaje del sistema operativo en el que se ejecuta la máquina virtual de Java.

7. ***AUTOMATIC_EVENTS_ENABLED***: este parámetro, permite habilitar o deshabilitar de manera manual, los eventos automáticos que genera el navegador como consecuencia de ciertas acciones (por ejemplo, el evento *onload* cuando finaliza la carga del documento HTML).

   Cuando los eventos automáticos están deshabilitados, el código JavaScript asociado a ellos se ignora y no se ejecuta.

8. ***SET_TIMEOUT_ENABLED***: permite establecer la política a seguir ante los eventos asíncronos, generados mediante temporizadores, establecidos con la función predefinida *setTimeout*.

   Con este parámetro, se pueden deshabilitar de forma íntegra, con lo que esos eventos se descartarían.

   También se puede permitir una sola ejecución del mismo contenido (cuando se establece un segundo temporizador, ejecutando el mismo contenido, se descarta), etc.

9. ***SET_INTERVAL_ENABLED***: permite establecer la política a seguir ante los eventos asíncronos, generados mediante temporizadores, establecidos con la función predefinida *setInterval*.

   Con este parámetro, se pueden deshabilitar de forma íntegra, con lo que esos eventos se descartarían.

   También se puede permitir una sola ejecución del mismo contenido (cuando se establece un segundo temporizador, ejecutando el mismo contenido, se descarta), etc.

10. ***ADVANCED_WINDOW_SCRIPTING***: este parámetro, permite deshabilitar de manera manual, ciertas funcionalidades JavaScript del objeto predefinido *window*.

    Este parámetro, es útil cuando se conoce de antemano que esas funcionalidades no se utilizan durante la evaluación de JavaScript.

    En esos casos, también se evita la construcción de determinadas estructuras internas (por ejemplo, índices), con la consiguiente mejora en el rendimiento del componente de navegación durante la ejecución de JavaScript.

11. ***ADVANCED_DOCUMENT_SCRIPTING***: este parámetro permite deshabilitar, de forma manual, ciertas funcionalidades JavaScript del objeto predefinido *document*.

    Este parámetro, es útil cuando se conoce de antemano que esas funcionalidades no se utilizan durante la evaluación de JavaScript.

En esos casos, también se evita la construcción de estructuras internas (por ejemplo, índices), con la consiguiente mejora en el rendimiento.

12. ***ADVANCED_NODE_SCRIPTING***: este parámetro permite deshabilitar, de forma manual, ciertas funcionalidades JavaScript de los nodos del árbol DOM.

    Este parámetro, es útil cuando se conoce de antemano que esas funcionalidades no se utilizan durante la evaluación de JavaScript.

    En esos casos, también se evita la construcción de algunas estructuras internas.

13. ***DOCUMENT_WRITE_ENABLED***: permite deshabilitar todos los contenidos dinámicos, generados con la función predefinida *write* del objeto *document*.

    Esta funcionalidad es muy útil en ciertos escenarios, cuando por medio de esta función, se genera contenido dinámico indeseado (por ejemplo *iframes* con publicidad), que no es necesario para el correcto funcionamiento de la secuencia de navegación web.

14. ***DOCUMENT_WRITE_DELAYED***: este parámetro, permite optimizar escenarios en los que el contenido dinámico generado con la función predefinida *write*, se hace a través de múltiples llamadas y, en cada una de ellas, se añade una pequeña porción del contenido final.

    Este escenario es bastante frecuente, y cuando se produce, se realizan múltiples llamadas a la función *write*, en lugar de una sola (es habitual que se haga así, sólo para mejorar la visualización del código fuente JavaScript en el editor de textos, durante el desarrollo de los contenidos del sitio web).

    Con este parámetro, todas las llamadas a esta función se retrasan y se agrupan, para realizar una sola invocación, una vez finalizada la evaluación del script en curso.

15. ***LIGHTWEIGHT_EVENTS_ENABLED***: este parámetro, permite optimizar la creación de objetos JavaScript de tipo evento (en este caso, no se refiere a eventos del motor de renderización, almacenables en la cola de eventos pendientes, sino que se refiere a objetos que se crean en el contexto de ejecución de JavaScript).

    Con este parámetro, la creación de eventos es más ligera y rápida, dado que no se inicializarán ciertos elementos avanzados, que se conoce de antemano que no se van a utilizar durante el procesamiento del manejador (por ejemplo, ante un evento de ratón, podría no realizarse el cálculo de las coordenadas).

16. ***JAVASCRIPT_INITIALIZATION_CODE***: permite definir código JavaScript de inicialización, que se ejecutará cada vez que se crea un nuevo navegador.

Con este código de inicialización, podrían establecerse funcionalidades adicionales, redefinir objetos, etc.

Este parámetro, es de gran utilidad cuando el sitio web utiliza alguna funcionalidad JavaScript, que todavía no está soportada de manera nativa en el componente de navegación.

17. ***ONLOAD_INITIALIZATION_CODE***: permite definir código JavaScript de inicialización, que se ejecutará cada vez que finaliza la carga de algún documento, dentro de alguna de las ventanas del navegador.

    Con este parámetro, se permite la inyección de JavaScript, con el objetivo de redefinir el comportamiento por defecto del componente de navegación, añadir alguna funcionalidad adicional que se utilizará más adelante, etc.

7.4 PARÁMETROS DE CONFIGURACIÓN DEL POOL DE *THREADS* DEDICADO A LAS DESCARGAS

En este apartado, se muestran los parámetros del componente de navegación, relacionados con el pool de *threads* dedicado a las descargas de los documentos de la red.

El sistema permite utilizar un pool por cada navegador, o se puede utilizar un solo pool, compartido por todos los navegadores que se ejecutan dentro de la misma máquina virtual.

1. ***MULTITHREAD_CONNECTION_MANAGER***: este parámetro, permite definir un gestor de descargas de un solo hilo o multihilo.

   En caso de utilizar un gestor de descargas multihilo, también es posible definir el número máximo de *threads* que podrá utilizar de manera simultánea.

2. ***MAX_HTTP_CONNECTION*S**: este parámetro, permite especificar el número máximo de conexiones HTTP que el gestor de descargas puede mantener abiertas de forma simultánea.

   Este número máximo, incluye las conexiones a todos los sitios web a los que están accediendo todos los componentes de navegación en ejecución.

3. ***MAX_HTTP_HOST_CONNECTIONS***: este parámetro, permite limitar el número de conexiones, por cada uno de los sitios web a los que están accediendo los componentes de navegación en un momento determinado.

4. ***CLOSE_HTTP_CONNECTIONS***: permite utilizar conexiones HTTP reutilizables o no reutilizables dentro del gestor de descargas.

   En caso de habilitar conexiones HTTP no reutilizables, éstas se cerrarán una vez finalizada la petición HTTP en curso.

5. ***GLOBAL_CONNECTION_MANAGER***: por medio de este parámetro, cada uno de los componentes de navegación que están en ejecución, podría utilizar una instancia del gestor de descargas diferente.

   Por defecto, todos los componentes de navegación utilizan el mismo gestor de conexiones (existe un gestor de conexiones global), pero en ciertos escenarios, puede ser conveniente utilizar una instancia diferente en algún componente de navegación. Por ejemplo, cuando un navegador acceda a un sitio web con información muy delicada, y se desea que este navegador se ejecute en un contexto lo más aislado posible.

   Otro escenario en el que un gestor dedicado puede ser de mucha utilidad, es cuando el tiempo de vida del navegador se prevé que sea muy largo y, por lo tanto, los requerimientos de este gestor de descargas, pueden ser diferentes a los que se establecen en la configuración por defecto (por

ejemplo, en este escenario, se podría utilizar un tiempo de vida mayor, para cada una de las conexiones TCP abiertas, etc.).

7.5 PARÁMETROS DE CONFIGURACIÓN DE LA CACHÉ DEL COMPONENTE DE NAVEGACIÓN

A continuación, se detallan los parámetros de configuración de la caché de ficheros persistentes y de la caché de scripts compilados.

Estas cachés, las comparten todos los navegadores que se ejecutan dentro de la misma máquina virtual Java.

1. ***CACHE_VALIDATION***: permite establecer la política de validación de las entradas en la caché de ficheros descargados.

   Por medio de este parámetro, se pueden establecer las siguientes políticas de validación:

   - Por tiempo máximo de vida.
   - Por tiempo máximo sin accesos (inactividad).
   - O incluso se puede establecer una política de validación más estricta en la que se ejecutan peticiones HTTP de tipo HEAD, que comprueban si el fichero ha cambiado en el servidor web.

   Para validar las entradas de la caché, aplicando la política establecida, existe un hilo dentro del componente de navegación dedicado de forma exclusiva a esta tarea.

2. ***CACHE_SIZE***: permite establecer el número máximo de entradas dentro de la caché de ficheros descargados.

   Cuando se alcanza este número máximo, la política para descartar elementos, también se puede configurar de entre las siguientes:

   - Descarte por número de accesos (se descarta la entrada menos usada).
   - Descarte por tiempo de vida (se descarta la entrada más antigua).

3. ***CACHE_TIME_TO_LIVE***: en caso de establecerse una política de validación por tiempo máximo de vida, mediante este parámetro se puede modificar el valor por defecto.

4. ***CACHE_CHECK_INTERVAL***: para la validación de las entradas de la caché de ficheros descargados, se utiliza un hilo dedicado dentro del componente de navegación. Mediante este parámetro, se puede establecer el tiempo de espera entre dos chequeos consecutivos.

5. ***SCRIPT_CACHE_SIZE***: tamaño máximo de la caché JavaScript compilado.

   Cuando se alcanza este tamaño máximo, la política para descartar entradas, también puede establecerse por tiempo o por número de accesos.

6. ***SCRIPT_CACHE_TIME_TO_LIVE***: tiempo máximo de vida de un script compilado en la caché.
7. ***SCRIPT_CACHE_CHECK_INTERVAL***: para la validación de las entradas en la caché de scripts compilados, existe otro hilo diferente dedicado en exclusiva a esta tarea. Con este parámetro, se puede configurar el tiempo de espera entre dos ejecuciones consecutivas del proceso de validación.

8 ANEXO II: PRUEBAS DE UNIDAD DEL COMPONENTE DE NAVEGACIÓN CON LAS PRINCIPALES LIBRERÍAS JAVASCRIPT

La implementación de referencia de la arquitectura del componente de navegación que ha sido desarrollado en esta tesis doctoral, da soporte a las librerías JavaScript, de libre distribución, más utilizadas hoy en día en sitios web reales de todo tipo de ámbitos.

Entre estas librerías JavaScript, podemos destacar *YUI* [YUI], *Dojo Toolkit* [DOJO] o *JQuery* [JQUERY], entre otras.

Algunas de estas librerías, proporcionan una gran batería de pruebas de unidad que son utilizadas por los propios desarrolladores de los diferentes proyectos, para garantizar que cada vez que se libera una nueva versión, el funcionamiento de la librería sigue siendo correcto en todos los navegadores convencionales.

De este modo, todas estas baterías de pruebas han sido utilizadas a lo largo de este proyecto, para conseguir que la implementación de referencia del componente de navegación, alcance la madurez suficiente para emular al navegador Microsoft Internet Explorer.

A continuación, se muestra el resultado de ejecutar las pruebas de unidad de las principales librerías, utilizando la implementación de referencia de la arquitectura del componente de navegación:

1. Dojo Toolkit [DOJO]: librería modular de código abierto, mantenida por la Dojo Foundation.

   Esta librería está formada por un módulo principal y una serie de módulos auxiliares, en los que se incluyen funcionalidades complementarias.

2. JQuery [JQUERY]: librería de código abierto que se ha ido convirtiendo, poco a poco, en una de las librerías más populares y utilizadas a día de hoy, en todo tipo de aplicaciones web.

   Esta librería, proporciona una serie de utilidades (mediante un API de alto nivel), para realizar diferentes operaciones sobre el árbol DOM de los documentos HTML.

3. Mochikit [MOCHI]: librería ligera y también de código abierto, inspirada en ideas de Python y de Objetive-C.

   Esta librería, proporciona entre otras funcionalidades, un mecanismo de ejecución en diferido, para simular comportamiento asíncrono.

4. Prototype [PROTO]: librería de código abierto que proporciona una serie de utilidades para desarrollar aplicaciones JavaScript.

   Esta librería, utiliza la aproximación de extender las funcionalidades predefinidas de los distintos objetos del árbol DOM.

5. Script-Aculo-Us [SCACUS]: librería JavaScript basada en Prototype que proporciona funcionalidades de más alto nivel, como por ejemplo, la generación de efectos visuales o la manipulación de la interfaz de usuario.

6. YUI [YUI]: librería JavaScript desarrollada por la empresa Yahoo!.

   En estos momentos, esta librería se encuentra discontinuada del desarrollo de nuevas funcionalidades, y tan solo se proporcionan nuevas versiones que solucionan problemas críticos.

Cabe destacar que aunque el porcentaje de éxito en la ejecución de pruebas de unidad es bastante elevado (como veremos a continuación, varía desde el 88% en el peor de los casos, hasta el 100% de éxito en el mejor de los casos), algunos de los errores no se deben a problemas reales en la implementación de la arquitectura de referencia.

En primer lugar, algunas pruebas no funcionan con éxito, en ninguno de los navegadores convencionales.

Por otro lado, muchos de los tests en los que se obtienen valores distintos a los que presentan los navegadores convencionales, es a causa de pequeñas variaciones en los cálculos de los atributos de visualización (por ejemplo, diferencias en el tamaño de los rectángulos), que no suponen un problema real a la hora de ejecutar secuencias de navegación sobre los sitios web.

8.1 PRUEBAS DE COMPATIBILIDAD CON LA LIBRERÍA DOJO

En primer lugar, la Tabla 13 muestra el resultado de la ejecución de las pruebas de unidad incluidas dentro de la librería *Dojo Toolkit* (versión 1).

En la tabla, se puede observar como la implementación de referencia de la arquitectura, ejecuta con éxito el 95% del total de 1680 pruebas.

A continuación, la Tabla 14 y la Tabla 15 (se ha dividido en dos tablas diferentes, para mejorar la visualización de los resultados), muestran el resultado de ejecutar las pruebas de unidad de la misma librería, pero en este caso, utilizando una versión más moderna (la versión 1.4).

En la tabla, se puede observar como la implementación de referencia de la arquitectura, ejecuta con éxito el 92% del total de 2478 pruebas.

	TESTS	SUCCESS	ERRORS / FAILURES
DOJO.TESTS.MODULE	674	631	14 / 29
TESTS.DATA	165	165	0 / 0
DIJIT.TESTS.MODULE	81	65	0 / 16
DOJOX.DTL.TESTS.MODULE	90	89	0 / 1
DOJOX.DATA.TESTS.MODULE	274	272	0 / 2
DOJOX.STRING.TESTS.STRING	16	16	0 / 0
DOJOX.DATE.TESTS.MODULE	5	4	1 / 0
DOJOX.WIRE.TESTS.MODULE	42	42	0 / 0
DOJOX.GFX.TESTS.MODULE	49	49	0 / 0
DOJOX.COLLECTIONS.TESTS.COLLECTIONS	75	75	0 / 0
DOJOX.VALIDATE.TESTS.MODULE	18	14	0 / 4
DOJOX.RPC.TESTS.MODULE	22	3	19 / 0
DOJOX.COLOR.TESTS.COLOR	9	9	0 / 0
DOJOX.UUID.TESTS.UUID	6	6	0 / 0
DOJOX.JSONPATH.TESTS.MODULE	19	19	0 / 0
DOJOX.CHARTING.TESTS.CHARTING	3	3	0 / 0
DOJOX.LANG.TESTS.MAIN	98	98	0 / 0
DOJOX.ENCODING.TESTS.CRYPTO	3	3	0 / 0
DOJOX.ENCODING.TESTS.COMPRESSION	8	8	0 / 0
DOJOX.ENCODING.TESTS.DIGESTS	3	3	0 / 0
DOJOX.ENCODING.TESTS.ENCODING	20	20	0 / 0
TOTAL DOJO 1	1680	1594 (95%)	86 (5%)

Tabla 13. Pruebas de compatibilidad con la librería Dojo versión 1.

	TESTS	SUCCESS	ERRORS / FAILURES
TESTS.DATA	209	209	0 / 0
TESTS.BASE	480	443	1 / 36
TESTS.I18N	9	9	0 / 0
TESTS.BACK-HASH	1	1	0 / 0
TESTS.HASH	22	21	1 / 0
TESTS.CLDR	1	1	0 / 0
TESTS.DATE	28	28	0 / 0
TESTS.NUMBER	30	30	0 / 0
TESTS.CURRENCY	1	1	0 / 0
TESTS.ADAPTERREGISTRY	5	5	0 / 0
TESTS.IO.SCRIPT	5	5	0 / 0
TESTS.IO.IFRAME	7	5	2 / 0
TESTS.RPC	4	2	2 / 0
TESTS.REGEXP	1	1	0 / 0
TESTS.STRING	6	6	0 / 0
TESTS.BEHAVIOR	5	5	0 / 0
TESTS.PARSER	23	23	0 / 0
TESTS.COLORS	26	26	0 / 0
TESTS.COOKIE	5	5	0 / 0
TESTS.FX	29	27	2 / 0
TESTS.DEFERREDLIST	8	8	0 / 0
TESTS.HTML	18	18	0 / 0
TESTS.NODELIST-TRAVERSE	15	15	0 / 0
TESTS.NODELIST-MANIPULATE	20	20	0 / 0
TESTS.CACHE	1	1	0 / 0
DOJOX.FX.TESTS.BASE	3	3	0 / 0
DOJOX.ROBOT.TESTS.MODULE	3	1	2 / 0
DOJOX.XML.TESTS.MODULE	10	9	0 / 1
DOJOX.DTL.TESTS.MODULE	99	98	0 / 1
DOJOX.DATA.TESTS.MODULE	521	515	0 / 6
DOJOX.GRID.TESTS.ROBOT.MODULE	3	0	3 / 0
DOJOX.STRING.TESTS.STRING	23	20	0 / 3
DOJOX.DATE.TESTS.MODULE	26	25	1 / 0
DOJOX.WIRE.TESTS.MODULE	42	37	0 / 5
DOJOX.IMAGE.TESTS.MODULE	10	8	2 / 0
DOJOX.HTML.TESTS.MODULE	23	13	0 / 10
DOJOX.GFX.TESTS.MODULE	54	54	0 / 0
DOJOX.COLLECTIONS.TESTS.COLLECTIONS	75	75	0 / 0

Tabla 14. Pruebas de compatibilidad con la librería Dojo versión 1.4 (primera parte)

	TESTS	SUCCESS	ERRORS / FAILURES
DOJOX.VALIDATE.TESTS.MODULE	21	17	0 / 4
DOJOX.HIGHLIGHT.TESTS.MODULE	2	2	0 / 0
DOJOX.TESTING.TESTS.MODULE	17	17	0 / 0
DOJOX.RPC.TESTS.MODULE	21	3	18 / 0
DOJOX.COLOR.TESTS.COLOR	5	5	0 / 0
DOJOX.ATOM.TESTS.IO.MODULE	20	13	6 / 1
DOJOX.JSON.TESTS.MODULE	47	47	0 / 0
DOJOX.UUID.TESTS.UUID	6	6	0 / 0
DOJOX.JSONPATH.TESTS.MODULE	19	19	0 / 0
DOJOX.LANG.TESTS.MAIN	137	135	0 / 2
DOJOX.MATH.TESTS.MAIN	29	29	0 / 0
DOJOX.ENCODING.TESTS.CRYPTO	12	12	0 / 0
DOJOX.ENCODING.TESTS.COMPRESSION	8	8	0 / 0
DOJOX.ENCODING.TESTS.DIGESTS	6	6	0 / 0
DOJOX.ENCODING.TESTS.ENCODING	20	20	0 / 0
DIJIT.TESTS.BASE	75	51	17 / 7
DIJIT.TESTS.INFRASTRUCTURE	52	40	3 / 9
DIJIT.TESTS.GENERAL	11	0	11 / 0
DIJIT.TESTS.TREE.MODULE	18	15	3 / 0
DIJIT.TESTS.EDITOR.MODULE	10	0	10 / 0
DIJIT.TESTS.FORM.MODULE	15	0	15 / 0
DIJIT.TESTS.LAYOUT.MODULE	76	54	5 / 17
TOTAL (DOJO 1.4)	2478	2272 (92%)	206 (8%)

Tabla 15. Pruebas de compatibilidad con la librería Dojo versión 1.4 (segunda parte).

8.2 PRUEBAS DE COMPATIBILIDAD CON LA LIBRERÍA JQUERY

La Tabla 16, muestra el resultado de la ejecución de las pruebas de unidad de la librería *JQuery* (versión 1).

En este caso, el 100% de las 1137 pruebas se ejecutan con éxito, utilizando la implementación de referencia de la arquitectura del componente de navegación.

A continuación, la Tabla 17, muestra el resultado de la ejecución de las pruebas de unidad de la versión 2 de la librería *JQuery*.

En este caso, el 89%, de un total de 6182 pruebas de unidad, se ejecuta con éxito utilizando la implementación de referencia de la arquitectura.

	TESTS	SUCCESS	ERRORS / FAILURES
CORE	561	561	0
FX	291	291	0
DIMENSION	18	18	0
EVENT	113	113	0
SELECTOR	154	154	0
TOTAL (JQUERY 1)	1137	1137 (100%)	0 (0%)

Tabla 16. Pruebas de compatibilidad con la librería JQuery versión 1.

	TESTS	SUCCESS	ERRORS / FAILURES
CORE	463	450	13
CALLBACKS	941	941	0
DEFERRED	239	203	36
SUPPORT	17	13	4
DATA	391	390	1
QUEUE	47	47	0
ATTRIBUTES	539	500	39
EVENT	405	396	9
SELECTOR	204	196	8
TRAVESING	351	298	53
MANIPULATION	706	654	52
WRAP	99	90	9
CSS	259	192	67
SERIALIZE	31	27	4
AJAX	358	240	118
EFFECTS	577	530	47
OFFSET	412	257	155
DIMENSIONS	143	87	56
TOTAL (JQUERY 2)	6182	5511 (89%)	671 (11%)

Tabla 17. Pruebas de compatibilidad con la librería JQuery versión 2.

8.3 PRUEBAS DE COMPATIBILIDAD CON LA LIBRERÍA MOCHIKIT

La Tabla 18, muestra el resultado de la ejecución de las pruebas de unidad proporcionadas por la librería *Mochikit*.

En este caso, el 100% de un total de 610 pruebas, se ejecuta con éxito utilizando la implementación de referencia de la arquitectura del componente de navegación.

8.4 PRUEBAS DE COMPATIBILIDAD CON LA LIBRERÍA PROTOTYPE

La Tabla 19, muestra el resultado de la ejecución de las pruebas de unidad de la librería *Prototype* utilizando la implementación de referencia.

En la tabla, se puede observar como la implementación de referencia de la arquitectura, ejecuta con éxito el 100% del total de 229 pruebas.

	TESTS	SUCCESS	ERRORS / FAILURES
TEST_MOCHIKIT-ASYNC	45	45	0
TEST_MOCHIKIT-BASE	149	149	0
TEST_MOCHIKIT-DATETIME	22	22	0
TEST_MOCHIKIT-DOM	73	73	0
TEST_MOCHIKIT-FORMAT	53	53	0
TEST_MOCHIKIT-ITER	66	66	0
TEST_MOCHIKIT-LOGGING	27	27	0
TEST_MOCHIKIT-MOCHIKIT	4	4	0
TEST_MOCHIKIT-COLOR	81	81	0
TEST_MOCHIKIT-SIGNAL	63	63	0
TOTAL (MOCHIKIT)	610	610 (100%)	0 (0%)

Tabla 18. Pruebas de compatibilidad con la librería Mochikit.

	TESTS	SUCCESS	ERRORS / FAILURES
TOTAL (PROTOTYPE)	229	229 (100%)	0 (0%)

Tabla 19. Pruebas de compatibilidad con la librería Prototype.

8.5 PRUEBAS DE COMPATIBILIDAD CON LA LIBRERÍA SCRIPT-ACULO-US

La Tabla 20, muestra el resultado de la ejecución de las pruebas de unidad proporcionadas por la librería *Script-Aculo-Us*.

En este caso, la ejecución finaliza con un 91% de las pruebas ejecutadas con éxito (del total de 993), utilizando la implementación de referencia.

	TESTS	SUCCESS	ERRORS / FAILURES
AJAX_AUTOCOMPLETER_TEST	25	15	2 / 8
AJAX_INPLACEEDITOR_TEST	86	78	0 / 8
BDD_TEST	52	52	0 / 0
BUILDER_TEST	331	331	0 / 0
DRAGDROP_TEST	25	25	0 / 0
EFFECTS_TEST	102	102	0 / 0
ELEMENT_TEST	46	45	0 / 1
LOADING_TEST	3	3	0 / 0
POSITION_CLONE_TEST	54	12	0 / 42
SLIDER_TEST	137	111	0 / 26
SORTABLE_TEST	31	31	0 / 0
STRING_TEST	34	34	0 / 0
UNITTEST_TEST	67	67	0 / 0
TOTAL (SCRIPTACULOUS)	993	906 (91%)	79 (9%)

Tabla 20. Pruebas de compatibilidad con la librería Script-Aculo-Us.

8.6 PRUEBAS DE COMPATIBILIDAD CON LA LIBRERÍA YUI

A continuación, en la Tabla 21, se muestra el resultado de la ejecución de las pruebas de unidad incluidas en la librería YUI (versión 2).

En este caso, se han ejecutado de manera correcta un total de 562 pruebas de unidad, que representan el 96% del total de 588 pruebas.

Por último, en la Tabla 22 y en la Tabla 23 (se ha dividido en dos tablas diferentes, para mejorar la visualización de los resultados), se muestra el resultado de la ejecución de las pruebas de unidad incluidas en la librería YUI, en este caso en la versión 3.

En esta ocasión, el resultado de la ejecución de las 1275 pruebas, es de un 88% de acierto, al utilizar la implementación de referencia de la arquitectura del componente de navegación.

	TESTS	SUCCESS	ERRORS / FAILURES
ANIMATION	5	5	0
AUTOCOMPLETE	6	6	0
CALENDAR	36	34	2
COOKIE	125	125	0
CONFIG	8	8	0
DATASOURCE	11	11	0
DATEMATH	23	23	0
DOM	30	28	2
DRAGDROP	8	3	5
ELEMENT	5	5	0
IMAGELOADER	9	8	1
LOGGER	3	3	0
PROFILER	11	11	0
TABVIEW	6	6	0
YAHOO	19	19	0
TESTREPORTING	8	8	0
ASSERT	24	24	0
OBJECTASSERT	3	3	0
ARRAYASSERT	9	9	0
MOUSEEVENT	44	39	5
KEYEVENT	18	18	0
USERACTION	62	57	5
YUITEST	115	109	6
TOTAL (YUI 2)	588	562 (96%)	26 (4%)

Tabla 21. Pruebas de compatibilidad con la librería YUI (versión 2).

	TESTS	SUCCESS	ERRORS / FAILURES
ASYNC-QUEUE	18	18	0
TEST	1	1	0
ATTRIBUTE	108	108	0
CACHE	25	25	0
CLASSNAMEMANAGER	4	4	0
ARRAYLIST	17	17	0
COLLECTION	21	21	0
CONSOLE	28	28	0
CONSOLE-FILTERS	3	3	0
COOKIE	168	168	0
DATASCHEMA	21	16	5
DATE	11	5	6
DATE-COMPAT	7	4	3
NUMBER	13	13	0
XML	6	6	0
DD	27	15	12
DOM	20	15	5
SELECTOR	17	11	6
SELECTOR_NEW	222	175	47
DUMP	1	1	0
DELEGATE	13	13	0
EVENT-DOM	16	12	4
EVENT-SYNTHETIC	5	5	0
FOCUSBLUR	6	6	0
AOP	7	6	1
CUSTOMEVENT	19	18	1
EVENT_SEQUENCE	7	7	0
EVENT-SIMULATE	66	61	5
IMAGELOADER	11	9	2
JSON	108	101	7
NODE	39	24	15
FOCUSMANAGER	4	3	1
NODE-FOCUSMANAGER	4	3	1
OOP	6	5	1
PLUGINS	16	16	0
PROFILER	14	14	0
QUERYSTRING	11	6	5
QUEUE	18	17	1
SLIDER	28	22	6

Tabla 22. Pruebas de compatibilidad con la librería YUI versión 3 (primera parte).

	TESTS	SUCCESS	ERRORS / FAILURES
STYLESHEET	26	10	16
SUBSTITUTE	1	1	0
ARRAYASSERT	18	18	0
MOCK	48	48	0
OBJECTASSERT	17	15	2
TESTFORMAT	3	3	0
WIDGET	10	10	0
PARENT-CHILD	8	8	0
ARRAY	4	3	1
CORE	1	1	0
OBJECT	3	3	0
TOTAL (YUI 3 PARTE 2)	1275	1122 (88%)	153 (12%)

Tabla 23. Pruebas de compatibilidad con la librería YUI versión 3 (segunda parte).

9 REFERENCIAS

[APCOMM] Apache commons. http://commons.apache.org.

[APHTTPC3] Apache commons HttpClient 3. http://hc.apache.org/httpclient-3.x/index.html.

[APHTTPC4] Apache HttpComponents. http://hc.apache.org.

[AFKL00] V. Anupam, J. Freire, B. Kumar, D. F. Lieuwen. Automating Web navigation with the WebVCR, Computer Networks 33(1-6):503-517, 2000.

[ALEXA] Alexa: The Web Information Company. http://www.alexa.com.

[APPSAF] Safari Web Browser. https://www.apple.com/safari.

[BHNV10] C. Badea, M. R. Haghighat, A. Nicolau, and A. V. Veidenbaum. Towards parallelizing the layout engine of firefox. In Proceedings of the Second USENIX Workshop on Hot topics in Parallelism, HotPar, pages 1–6, 2010.

[CSPWM13] Calin Cascaval, Seth Fowler, Pablo Montesinos-Ortego, Wayne Piekarski, Mehrdad Reshadi, Behnam Robatmili, Michael Weber, and Vrajesh Bhavsar. 2013. ZOOMM: a parallel web browser engine for multicore mobile devices. In Proceedings of the 18th ACM SIGPLAN symposium on Principles and practice of parallel programming (PPoPP '13). ACM, New York, NY, USA, 271-280.

[CSS1] CSS Level 1 Specification. http://www.w3.org/TR/REC-CSS1.

[CSS2] CSS Level 2 Specification. http://www.w3.org/TR/REC-CSS2.

[CSS3] Cascading Style Sheets (CSS) Snapshot 2010. http://www.w3.org/TR/CSS/.

[CSSPAR] CSS Parser. http://cssparser.sourceforge.net.

[DENODO] Denodo Technologies. www.denodo.com.

[DNDITP] Denodo ITPilot. http://www.denodo.com/en/solutions/horizontal-solutions/web-automation.

[DNDPLATF] Denodo Platform. http://www.denodo.com/en/denodo-platform/overview.

[DOJO] Dojo ToolKit. http://dojotoolkit.org.

[DYKR00] Davulcu H., Yang G., Kifer M., Ramakrishnan I.V.: Computational Aspects of Resilient Data Extraction from Semistructured Sources, ACM Symposium on Principles of Database Systems (PODS), pp. 136–144 (2000).

[DOM] Document Object Model (DOM). http://www.w3.org/DOM.

[ECMASC] ECMAScript Language Specification 262 3rd edition. http://www.ecma-international.org/publications/files/ECMA-ST-ARCH/ECMA-262, 3rd edition, December 1999.pdf.

[ELEFF] Proyecto Electrolysis Firefox. https://wiki.mozilla.org/Electrolysis.

[ENVJS] EnvJS. http://www.envjs.com.

[FTP] The Internet Society. File Transfer Protocol. https://www.ietf.org/rfc/rfc959.txt.

[GOOCHR] Google Chrome Web Browser. https://www.google.com/chrome.

[GOOBLK] The Chromium Project. Blink. http://www.chromium.org/blink.

[GOOSS] Google Chrome Script Streaming. http://blog.chromium.org/2015/03/new-javascript-techniques-for-rapid.html.

[GG05] Grosskurth, A., Godfrey, M. W., September 2005. A reference architecture for web browsers. In: ICSM'05: Proceedings of the 21st IEEE International Conference on Software Maintenance (ICSM'05). pp. 661–664.

[HM07] Hupp D., Miller R.C.: Smart Bookmarks: automatic retroactive macro recording on the web. In: Proceedings of the 20th Annual ACM Symposium on User Interface Software and Technology, pp. 81–90. ACM New York, Newport (2007).

[HSSCP12] H. Mai, S. Tang, S. T. King, C. Cascaval, and P. Montesinos. A case for parallelizing web pages. In Proceedings of the 4th USENIX conference on Hot Topics in Parallelism, HotPar'12, Berkeley, CA, USA, June 2012. USENIX Association.

[HTMUNIT] HtmlUnit. http://htmlunit.sourceforge.net.

[HTTP11] The Internet Society. Hypertext Transfer Protocol 1.1. http://tools.ietf.org/html/rfc2616.

[IMACROS] iOpus. iMacros. http://www.iopus.com.

[JAUNT] Jaunt Java Web Scraping & Automation. http://jaunt-api.com.

[JQUERY] JQuery. http://www.jquery.com.

[JSDOM] Jsdom. A JavaScript implementation of the DOM and HTML standards. https://www.npmjs.com/package/jsdom.

[KAPOW] Kapow: http://www.openkapow.com.

[LE07] Lingam S., Elbaum S.: Supporting End-Users in the Creation of Dependable Web Clips. WWW 2007, 953–962.

[LRPM12] Losada J., Raposo J., Pan A., Montoto P.: Efficient execution of web navigation sequences. In: The 14th International Conference on Web Information System Engineering (WISE), pp. 340–353 (2012).

[MOCHI] Mochikit. http://mochi.github.io/mochikit.

[MOZENDA] Mozenda. http://www.mozenda.com.

[MOZFF] Mozilla Firefox Web Browser. https://www.mozilla.org/firefox.

[MOZGCK] Mozilla Developer Network. Mozilla Gecko. https://developer.mozilla.org/en-US/docs/Mozilla/Gecko.

[MOZRHN] Mozilla Developer Network. Mozilla Rhino. https://developer.mozilla.org/en-US/docs/Mozilla/Projects/Rhino

[MOZSPD] Mozilla Developer Network. SpiderMonkey. https://developer.mozilla.org/en-US/docs/Mozilla/Projects/SpiderMonkey.

[MOZSPPAR] Mozilla Developer Network. Optimizing your pages for speculative parsing. https://developer.mozilla.org/en-US/docs/HTML/Optimizing_Your_Pages_for_Speculative_Parsing, 2012.

[MPBL11] Montoto, P., Pan, A., Raposo, J., Bellas, F., López, J.: Automated browsing in AJAX websites. Data Knowl. Eng. 70(3), 269–283 (2011).

[MSIE] Internet Explorer web browser. http://windows.microsoft.com/en-us/internet-explorer.

[MSTRDT] Internet Explorer Architecture. http://msdn.microsoft.com/en-us/library/aa741312(v=vs.85).aspx

[NEHTML] CyberNeko HTML Parser. http://nekohtml.sourceforge.net.

[NODEJS] NodeJS. https://nodejs.org.

[OPERA] Opera web browser. http://www.opera.com.

[PHAJS] PhantomJS. http://www.phantomjs.org.

[PRAHV02] A. Pan, J. Raposo, M. Álvarez, J. Hidalgo, A. Viña. Semi-Automatic Wrapper Generation for Commercial Web Sources. Proceedings of IFIP WG8.1 Working Conference on Engineering Information Systems in the Internet Context (EISIC), 2002, pp 265-283.

[PROTO] Prototype. http://prototypejs.org.

[QEGN] QEngine. http://www.adventnet.com/products/qengine/index.html.

[RPAHV02] J. Raposo, A. Pan, M. Álvarez, J. Hidalgo, Á. Viña. The Wargo System: Semi-Automatic Wrapper Generation in presence of Complex Data Access. Proceedings of the 13th International Workshop on Database and Expert Systems Applications (DEXA 2002). ISSN: 1529-4188. ISBNs: 0-7695-1668-8, 0-7695-1669-6 (case), 0-7695-1670-X (microfiche). Páginas: 313-317. Editorial: IEEE Computer Society Press. (2002).

[SAHI] Sahi. http://sahi.co.in/w.

[SAXAPI] Simple API for XML. http://sax.sourceforge.net/.

[SCACUS] Script-Aculo-Us. https://script.aculo.us.

[SIMBRO] SimpleBrowser. https://github.com/axefrog/SimpleBrowser.

[SKC01] Safonov A., Konstan J., Carlis J.: Beyond hard-to-reach pages: interactive, parametric web macros. In: 7th Conference on Human Factors & the Web. Madison 2001.

[SLNM] Selenium: http://seleniumhq.org.

[SPDY] SPDY Protocol - Draft 3.1. https://www.chromium.org/spdy/spdy-protocol/spdy-protocol-draft3-1.

[TWILL] Twill: a simple scripting language for Web browsing. http://twill.idyll.org.

[V8] V8 JavaScript Engine. https://code.google.com/p/v8.

[W3CDOM] The W3 Consortium. The Document Object Model. http://www.w3.org/DOM.

[W3CDOMP] The W3 Consortium. DOM Parsing and Serialization. http://www.w3.org/TR/DOM-Parsing.

[W3CHTML4] The W3 Consortium. HTML 4.01 Specification. http://www.w3.org/TR/html4.

[W3CHTML5] The W3 Consortium. HTML 5 Specification. http://www.w3.org/TR/html5.

[W3CRDF] The W3 Consortium. RDF/XML Syntax Specification. http://www.w3.org/TR/REC-rdf-syntax.

[W3CSVG] The W3 Consortium. Scalable Vector Graphics (SVG) 1.1 (Second Edition). http://www.w3.org/TR/SVG/.

[W3CWW] The W3 Consortium. Web Workers. http://www.w3.org/TR/workers.

[W3CXML] The W3 Consortium. Extensible Markup Language. http://www.w3.org/TR/REC-xml.

[W3CXHTML] The W3 Consortium. XHTML™ 1.0 The Extensible HyperText Markup Language. http://www.w3.org/TR/xhtml1.

[W3CXPT] The W3 Consortium. XML Path Language (XPath). http://www.w3.org/TR/xpath.

[WEBGL] WebGL Specification. https://www.khronos.org/registry/webgl/specs/1.0.

[WEBKIT] The WebKit Open Source Project. http://www.webkit.org.

[XMLHTTP] XMLHttpRequest Level 1. W3C Working Draft. http://www.w3.org/TR/XMLHttpRequest.

[XPATH] XML Path Language (XPath). http://www.w3.org/TR/xpath.

[YUI] Yahoo User Interface Library. http://yuilibrary.com.

[ZOMJS] Zombie.js. http://zombie.labnotes.org.